DC Power Flow Based Contingency Analysis Using Graphics Processing Units

A Thesis

Submitted to the Faculty

of

Drexel University

by

Anupam Gopal

In partial fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical Engineering

March 2010

ACKNOWLEDGMENTS

Current engineering advances are possible due to amalgamation of various technologies, which develop in isolation, but find interesting implementation when exploited to their limits. Using graphics processing units for contingency analysis is one such effort. One of the major roadblocks in pursuing such an endeavor is to gain expertise and insight in both these fields simultaneously. I was fortunate enough to have the support and guidance of two people who are experts in their respective fields.

I would like to thank Dr. Dagmar Niebur for offering her expertise in the field of power system analysis and Dr. Suresh Venkat Subramanian for his guidance in the field of graphics computing. Without their help and guidance, it was next to impossible to have this thesis in pen and paper.

I consider myself extremely lucky to have these remarkable people to help me out in making this thesis possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

This thesis explores the possibility of mapping power flow algorithms on a graphics processor. In particular we demonstrate the implementation of DC power flow based contingency analysis on a graphic processing unit (GPU).

GPU's are SIMD processors with highly streamlined architecture to support rendering of graphic images on the computer screen. However, in the recent decade, there has been great interest in exploiting the GPU architecture for purposes beyond rendering of graphic images. Mapping power flow algorithm to this processor is one such attempt.

Contingency analysis for power systems is a computationally exhaustive process, but at the same time is a valuable exercise for power system engineers in order to ensure power system security under various operating conditions. Numerous previous attempts have been made to efficiently implement the contingency algorithm on various high performance architectures, however there has been no attempt to implement the algorithm on a Graphics processor. There are numerous advantages to such an implementation, which is outlined in the thesis.

In particular following has been accomplished as part of the research work leading up to this thesis.

1. Modeling the DC power flow based contingency analysis algorithm to align with the highly pipelined architecture of GPU.

2. Optimizing the algorithm to enable efficient use of the GPU architecture and exploiting the inherent sparsity of Power system matrices.
3. Implementation of the algorithm.
4. Analysis of the speedup achieved as a result of change in system size.

In essence we have demonstrated that power system algorithms could be successfully implemented on the GPU's. Further research would be required to take advantage of the advances in the graphics computing industry, along with further streamlining of the algorithm to achieve higher throughput.

DC Power Flow Based Contingency Analysis Using Graphics Processing Units

Anupam Gopal
Dagmar Niebur, Ph.D.

1 INTRODUCTION

As we move towards a more industrialized society, our dependency on reliable energy supply grows tremendously. There is a growing need to have a more reliable source of energy, which can serve our increased demand and dependency. This puts the onus on the power system engineer to simulate all unforeseen conditions in advance and be better prepared to handle these contingencies.

Contingency Analysis for large-scale power systems has been known to be notoriously computationally intensive. One of the major problems faced by power system engineers across utilities is the lack of computing power while simulating these contingencies. As most of the utilities operate under strict financial constraints, access to high performance computing facilities is rather limited. With the growing need, for reliability, it becomes imperative for the engineers to run these resource-intensive simulations.

1.1 Power System Steady State Simulation

Power system simulation typically tends to involve solutions to a set of algebraic non linear equations for steady state analysis or a set of first order differential equations in conjunction with algebraic equations for transient stability. Using implicit integration techniques set of implicit non-linear algebraic equations needs to be solved at each integration time-step. Solution of these equations with a Newton-Raphson-type algorithm involves the inversion of a large and usually sparse matrix.

For a power system with *N* buses, the steady state model of the power system is given by a set of up to 2*N*-2 nonlinear equations, which are iteratively, solved using a Newton-Raphson-type algorithm. For a system affected by the blackout of the North-eastern Interconnection in August 2003, the number of buses *N* is in the order of 43,000 and the number of potential line losses is in the order 56,000. To successfully compute the power flow in the phase domain in real time for these types of systems requires high computational throughput.

It was realized early on that there is a certain degree of parallelism that could be exploited, for the purpose of faster execution of the solution algorithms. Most approaches fall into two distinct categories. One approach was focused on leveraging the advancement in the hardware technology, while the other approach was to manipulate the algorithm to make it more and more parallelizable. These early approaches however were less dedicated to optimizing the cost of resources dedicated for parallel computing.

In the late 90's there were several attempts made to exploit the sparsity in conjunction with parallelism of the power system equations of the form **Ax=B**. As indicated in [10], the success of parallelization depends on three factors: the problem structure, the computer architecture, and the algorithm that takes maximum advantage of both. There were several attempts like [11] [13] to come up with new parallel algorithms for matrix inversion like *Very Dishonest Newton Method* (VDHN) and *Shifted-Picard* (SP)[15]. Like-wise there had been multiple approaches to experiment with various multiprocessor architectures like the data sharing hypercube etc. [10] [12].

Most of these attempts gauged the performance of the modified algorithm, with the increase in the number of processors and with change in the parallel architecture of the processors. In other words researchers were addressing scalability and speed of the algorithms with the increase in system size, number of computation nodes, and the architecture of the arrangement of the nodes. These attempts were for the most part focused on shared/distributed memory multiprocessor architecture, with message passing interface serving as the communication backbone.

Previous approaches to parallelize the computation using various high performance computing platforms involved superscalar processors, vector processors, shared memory multiprocessors, distributed memory multi-computers and heterogeneous network of workstations [1-2]. Often these had to be programmed using assembly language or insufficiently tested cross-compilers. These previous efforts could not stand the test of time due to prohibitive cost involved in commercializing this type of specialized hardware. Most recently RTDS, a power system time-domain simulation system has gained some interest, [6]. While this system is highly suited for hardware-in-the-loop testing, for example digital relays, it was not developed for large-scale power system simulation. According to the manufacturer the largest power system simulated so far is a 300-bus KEPCO system.

1.2 Computer Graphic Processing Units

On the other hand commodity computer graphics processing units (GPU) are probably today's most powerful computational hardware per dollar installed on general purpose

PC, [3]. Already in the early 2000 the graphics processors performance improved dramatically; there was almost a 17-fold increase in computation horsepower when compared in gigaflops. In early 2003 the computation power was close to 10 gigaflops, while in 2006 it was as high as 170 gigaflops. Today's GPUs can process over 50 million triangles and 4 billion pixels in one second. Their “Moore's Law” is faster than that for CPUs, owing primarily to their stream architecture which enables all additional transistors to be devoted to increasing computational power directly. An intriguing side effect of this is the growing use of a graphics card as a general-purpose stream processing engine. In an ever-increasing array of applications, researchers are discovering that performing a computation on a graphics card is far faster than performing it on a CPU, and thus they are using GPUs as a stream co-processor. Another feature that makes the graphics pipeline attractive (and distinguishes it from other stream architectures) is the spatial parallelism it provides. Conceptually, each pixel on the screen can be viewed as a stream processor, potentially giving a large degree of parallelism.

Programming languages like “Cg” or “Brooks GPU” provide a high-level programming environment including graphics libraries. This is in contrast to many earlier dedicated processors that required assembly language coding.

Despite the increase in computational horse power the price of the processor remained quite low, due to huge requirement in the gaming market. On the other hand the increase in the performance of Pentium and other general-purpose processor was quite modest. This was due to the fundamental architectural difference between the CPU and GPU. The

CPU was optimized for high performance on sequential code, while GPU was optimized for graphics computation which is highly parallel in nature.

It is precisely during this time when there was a sudden interest in the engineering community about the possibility of porting general-purpose engineering algorithms onto GPU, which is highly optimized for graphics computation. There was quite a popular initiative called GPGPU (General Purpose Computation on Graphics Processing Unit), which focused entirely upon implementing general-purpose computation on the GPU. Most of these early initiatives were predominantly quite popular in the mechanical engineering community.

Our paper [14], which reports some results of this thesis, was the first attempt to port a power system application on the graphics processor. Given the highly parallel nature of the graphics processor, and the inherent parallel nature of the contingency processing algorithm, it was quite natural to port the algorithm onto this new architecture. The fact that the contingency algorithm is so inherently parallel, and the result of one contingency has no relation to other contingencies, makes it an ideal candidate for such an architecture without the overhead of extra cost/ complexity of large clusters/supercomputers.

In this thesis we present an implementation of a DC Power flow based contingency analysis, using Gauss-Jacobi iterations, on a Graphics Processing Unit. We then compare the performance of the algorithm on the GPU with its implementation on the CPU.

1.3 Organization of Thesis

This thesis can be broadly subdivided into 4 broad sections.

The first two chapters deal broadly with fundamentals of DC power flow based contingency analysis, and introduces the Graphic Processing Units. The introductory chapters assume some basic understanding of power systems and graphic processors. The chapters are in no way exhaustive, additional reading is required to get a complete understanding.

The second section pertains to the implementation details of the algorithm. It discusses the assumptions and design choices made, while implementing the algorithm on the GPU.

The third section presents a sample four-bus system, and explains the steps when the algorithm is executed.

The fourth section presents the simulation results of the implementation of the algorithm for standard systems. It also provides a brief analysis of the results of the simulation and suggests future work that can be done to advance this approach.

2 DC POWER FLOW BASED CONTINGENCY ANALYSIS

2.1 Overview

The complex power flow equations can be represented as

$$\mathbf{S}_i = \mathbf{V}_i \sum_{k=1}^{n} \mathbf{Y}_{ik}^{*} \mathbf{V}_k^{*} \tag{1}$$

Here $\mathbf{S}_i$ is complex power injection at bus i, $\mathbf{Y}_{ij}$ is admittance between bus i and j, and n is the number of busses. $\mathbf{V}i$ and $\mathbf{Y}_{ij}$ can be expanded as

$$\begin{aligned} \mathbf{V}_i &= \left|\mathbf{V}_i\right|\left(\cos\delta_i + j\sin\delta_i\right) \\ \mathbf{Y}_{ij} &= \left|\mathbf{Y}_{ij}\right|\left(\cos\theta_{ij} + j\sin\theta_{ij}\right) \end{aligned} \tag{2}$$

real part yields the active power equation for the active power injection $\mathbf{P}_i$ at bus i

$$\mathbf{P}_i = \sum_{k=1}^{n} \left|\mathbf{V}_i\right|\left|\mathbf{V}_k\right|\left|\mathbf{Y}_{ik}\right|\cos\left(\theta_{ik} + \delta_k - \delta_i\right) \tag{3}$$

The states of the system to be calculated are the voltage angles δ_i and the voltage magnitudes $\left|\mathbf{V}_i\right|$. We can represent (3) in a more compact form as

$$\mathbf{P} = f_P\left(\left|\mathbf{V}\right|, \delta\right) \tag{4}$$

Here **P** is the active power injection vector at load busses, and $|\mathbf{V}|$ is the voltage magnitude vector at the load buses and δ is the voltage phase angle vector at all buses except the slack bus. Taylor expansion of (4) around the operating point ($|\mathbf{V}_0|$, $\boldsymbol{\delta}_0$) gives

$$\mathbf{P} \approx f_P\left(|\mathbf{V}_0|, \boldsymbol{\delta}_0\right) + \frac{\partial f_P}{\partial |V|}\left(|\mathbf{V}| - |\mathbf{V}_0|\right) + \frac{\partial f_P}{\partial \delta}\left(\boldsymbol{\delta} - \boldsymbol{\delta}_0\right) \tag{5}$$

Because active power injections are only weakly coupled with the voltage magnitude, we can neglect the voltage sensitivities in the first equation and approximate the phase angle deviations and the reactive power deviations in (5) by

$$\left(\frac{\partial f_P}{\partial \delta}\right)^{-1}\left(\mathbf{P} - \mathbf{P}_0\right) \approx \left(\boldsymbol{\delta} - \boldsymbol{\delta}_0\right) \tag{6}$$

The linearized power flow model given in (6) can be further simplified by assuming a lossless system, thus neglecting the real parts of **Y** and by neglecting the voltage deviations. This model is the so called linear DC power flow model for the active power flow deviations. Under these assumptions (6) can be written as

$$\begin{aligned} \Delta\mathbf{P} &= \mathbf{B}\Delta\boldsymbol{\delta} \\ \mathbf{B} &= \left(\frac{\partial f_P}{\partial \delta}\right)\Big|_{r_{ij}=0} \end{aligned} \tag{7}$$

Here the variables are

B = Imag(Y) The bus reactance matrix (excluding the slack bus).

$\Delta\boldsymbol{\delta}$ The angle mismatch vector to be computed.

$\Delta\mathbf{P}$ The real power mismatch vector at each bus excluding the slack bus.

The above mentioned DC power flow is one of the most popular algorithms used for contingency analysis, due to reduced computational complexity involved and sufficient accuracy with respect to the megawatt flows on the line. In order to perform contingency analysis on a system, equation (7) is solved for various outage cases. A single branch outage affects 4 elements in the **B** matrix.

Then the DC power flow is solved for the contingent system. This process is repeated until all outages are successfully simulated. For large systems, outages are ranked according to certain performance indices and only a reduced number of contingencies are evaluated. 3.2 .Iterative Algorithms for System of Linear Equations

Solving the DC flow equations corresponds to solving a system of linear equations

$$\mathbf{Ax} = \mathbf{y} \tag{8}$$

Here **x** and **y** are the unknown and given real vectors of dimension N and **A** is an $N \times N$ real square matrix. In terms of the DC flow formulation in eqn. (7).

We thus have $\mathbf{A}=\mathbf{B}$, $\mathbf{x}=\Delta\mathbf{P}$, $\mathbf{y}=\Delta\boldsymbol{\delta}$ and $N=n-1$. .

Assuming that **A** is non-singular, a unique solution **x** to eqn.(8) exists.

Solution algorithms for the linear system include LU factorization, also known as Gauss-Jordan as well as the iterative algorithms including Gauss-Jacobi and Gauss-Seidel. It is a

well-known fact that the number of operations for the Gauss-Jordan type algorithms grows with N^3 whereas the number of operations grow with N^2 per iteration for the iterative algorithms. Iterative algorithms are thus of advantage for large systems of equations.

Furthermore, iterative algorithms exploit the inherent parallelism of the GPUs very efficiently. The following section briefly reviews the structure of two of these algorithms, [8].

2.1.1 Gauss-Jacobi Algorithm

The iterative mechanism for Gauss-Jacobi algorithm is represented in (9). It uses the values of x (i) at iteration i on the right side in(9). The kth component can be expressed as

$$x_k(i+1) = \frac{1}{A_{kk}}[y_k - \sum_{n=1}^{k-1} A_{kn}x_n(i) - \sum_{n=k+1}^{N} A_{kn}x_n(i)] \tag{9}$$

where k=1,…, N.

2.1.2 Gauss-Seidel Algorithm

Comparing the Gauss-Jacobi with Gauss-Seidel iteration technique, both the techniques are similar except for the fact that during each iteration, the updated values $x_n(i+1)$, for $n < k$. are used on the right side of (10) to update the values $x_k(i+1)$ on the left side. (10)

$$x_k(i+1) = \frac{1}{A_{kk}}\left(y_k - \sum_{n=1}^{k-1} A_{kn}x_n(i+1) - \sum_{n=k+1}^{N} A_{kn}x_n(i) \right) \tag{10}$$

In matrix form both algorithms can be formulated as

$$\mathbf{x}[i+1] = \mathbf{x}\left[i\right] - \mathbf{D}^{-1}\mathbf{A}\mathbf{x}\left[i\right] + \mathbf{D}^{-1}\mathbf{y} \tag{11}$$

For Gauss-Jacobi D is defined as

$$\mathbf{D} = \begin{pmatrix} A_{11} & & 0 \\ & \ddots & \\ 0 & & A_{nn} \end{pmatrix} \tag{12}$$

and for Gauss-Seidel D is defined as

$$\mathbf{D} = \begin{pmatrix} A_{11} & & 0 \\ \vdots & \ddots & \\ A_{n1} & \cdots & A_{nn} \end{pmatrix} \tag{13}$$

3 GRAPHICS PROCESSING UNITS

3.1 Overview

The arithmetic power of the GPU is a result of its highly specialized architecture, evolved over the years to extract the maximum performance on the highly parallel tasks of traditional computer graphics. Graphics tasks exhibit high degree of inherent parallelism, and hence can be programmed into kernels, contributing to the high arithmetic intensity [6].

The key to using the GPU for power flow computation is to view it as a streaming data parallel computer. In a stream model, programs are expressed as series of operations on data streams. The elements in a stream are processed by the instruction in a kernel. A kernel operates on each element of a stream and writes the result to an output stream. The stream elements that a card processes are points, and at a later stage, fragments, which are essentially pixels with color and depth information. A program issues requests to the card by specifying points, lines, or facets, (and their attributes) and issuing a rendering request. The vertices go through several processing phases (called a vertex program) and finally are scan-converted into fragments by a rasterizer that takes a three-dimensional scene and breaks it into pixels. Each fragment is then processed by another stream computation (called a fragment program) and finally ends up as a pixel on a screen. Alternatively, the pixels and their attributes can be captured in internal storage and their contents can be retrieved to the CPU. Spatial parallelism occurs at the level of a fragment; each fragment program can be viewed as running in parallel at each pixel.

Vertex programs are executed on each point. Texture memory provides a form of limited local storage, although access to it is restricted. [7].

3.2 Glossary of Graphics Computing Terms

A. *Textures*: The native data layout on CPU is one-dimensional array. Higher dimensional arrays defined in CPU computing are, inherently one-dimensional. The way the elements are accessed in these higher dimensional arrays is by offsetting the indices. In GPU the native data layout is two-dimensional. Data is stored in these two dimensional arrays called textures. This is essentially due to the fact that GPU's are designed to display two-dimensional geometry. Hence the hardware excels at manipulating two-dimensional data. The elements of these textures are accessed by indexes called *"texture co-ordinates"*.

B. *Kernels*: Kernels are the small fragment programs written into the fragment processor. These programs operate independently on the incoming data streams. Kernels can be compared to an inner loop of a double for loop, in terms of CPU style programming. These programs are the "Single Instructions" in the "Single Instruction Multiple Data (S.I.M.D)" style programming.

C. Render –to-Textures: In a CPU style procedural language programming it is easy to implement feedback, which is, output of a previous step serves as the input to the next step. This is because of the unified memory model where we could perform random read and writes, that is: textures could either be read or written at a time. So the result of the previous computation has to be written to a texture in a

write only mode. Once this step is over, the texture is converted into a read only mode, and is fed as input to the next step. This process of writing the result to the texture instead of rendering it to the screen is called render-to-texture.

D. *Rasterization:* Rasterization is the process by which we invoke computation. Once the fragment processors are programmed and textures are assigned, we need to invoke computation. Rendering a quadrilateral does this. In doing so:

   a. The vertex processor draws the geometry,
   b. Rasterizer determines the entire pixel covered by the geometry and generates corresponding fragments.
   c. Fragment processor performs math operations on each of the generated fragments in the input stream.
   d. Output is written to the frame buffer.

E. *Texture Coordinates:* Texture co-ordinates are equivalent to array indexes in CPU. The only difference being these co-ordinates address to four tuples of data as opposed to an array index referring to a single entity. These four tuples of data are the intensity of the colors (Red, Green, Blue and Opacity).

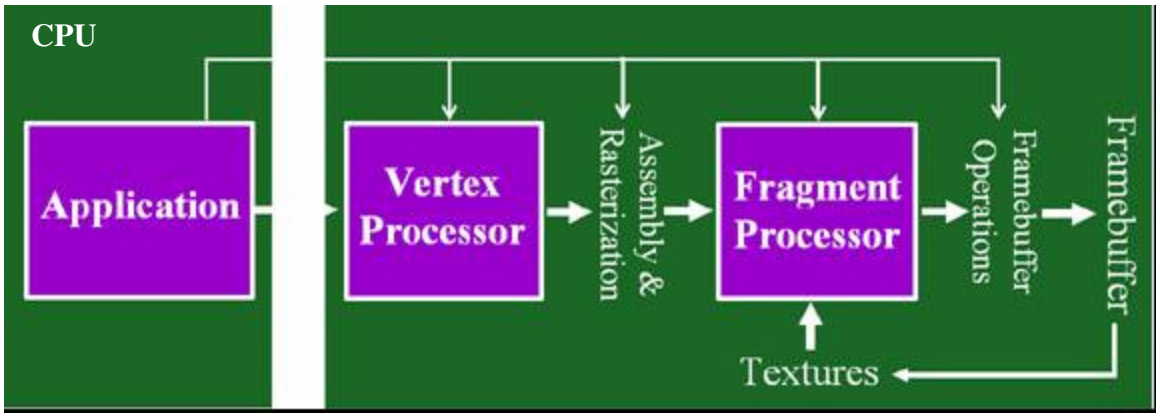
3.3 Graphics Hardware Pipeline

A pipeline is a sequence of stages operating in parallel and in a fixed order. Each stage receives its input from the prior stage and sends its output to the subsequent stage. Prior to 2002 vertex and fragment processors were not programmable, and hence a general purpose application could not be ported onto these processors. It was only after 2002 (4^{th} generation GPU's) that the two processing stages; namely, Vertex and fragment processing could be programmed. Rasterization stage is still hardwired, and is not accessible to a programmer. This level of programmability opens up the possibility of offloading complex vertex and pixel processing operations from CPU to GPU. In order to program these processors "Cg" (C for computer graphics) is used. Details about the language and its features would be explained in the next section.

The graphics hardware pipeline can be broadly divided into three stages from a general-purpose computation point of view as mentioned below:

- Vertex Processing.
- Primitive assembly and rasterization.
- Fragment Processing.

Additionally, the above mentioned stages could again be subdivided into various small stages, but from a power system computation point of view, it holds little significance. *Figure 1* shown below provides an overview of a modern graphics pipeline.



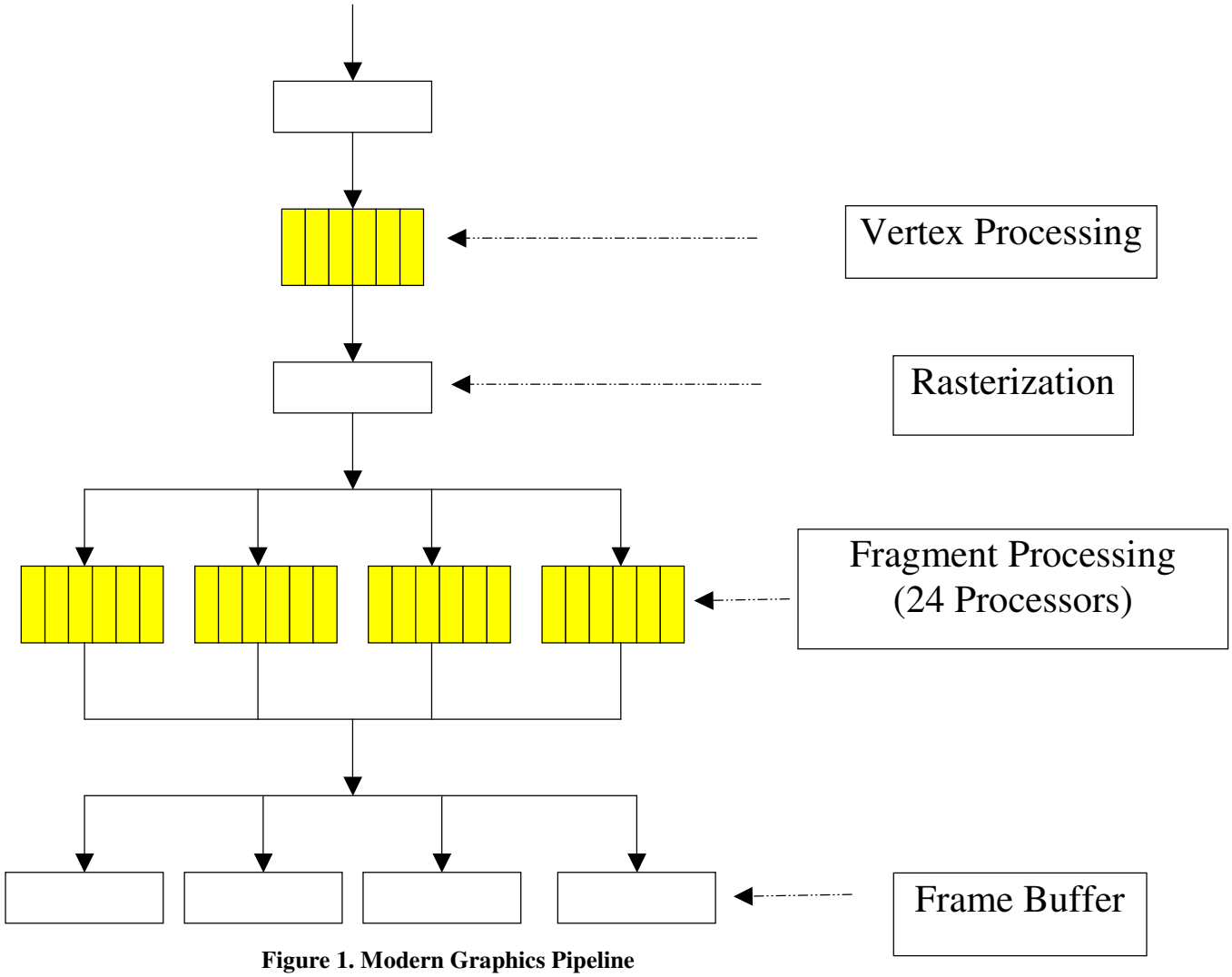


Figure 1. Modern Graphics Pipeline

Vertex Processing is the first processing stage in the graphics hardware pipeline. The vertex transformation performs a sequence of math operations on each vertex. These operations include transforming the vertex position into screen position for use by the rasterizer, generating texture coordinates for texturing, and lighting the vertex to determine its color.

The transformed vertices flow in sequence to the next stage, called primitive assembly and rasterization. The primitive assembly step assembles vertices into geometric primitives (triangles, lines or points). Hence rasterization can be described as a process of determining the set of pixels covered by these set of geometric primitives.

Once a primitive is rasterized into a collection of fragments, the fragment processing stage performs a sequence of texturing and math operations, and determines a final color for each fragment. The term fragment is used because this processor operates on each pixel data associated with the primitive, as determined by the rasterizer. Once the fragments are processed they are sent to the frame buffer for final display onto the screen.

3.4 Programming Language

There are two programmable processors in GPU that require to be programmed in order to implement a general-purpose application. We have used “Cg” as our programming language to implement the contingency analysis algorithm. “Cg” stands for “C in computer graphics”. Cg provides us with a programming language and a compiler to transform the vertex and fragment program into machine language. One of the main

reasons for using this language is due to its similarity to C. Moreover Cg also provides us with an added advantage of speed over other high level languages like “Brooks”.

“Cg” has been designed on a data flow model in contrast to C, which is basically a general-purpose procedural language. In Cg computation occurs in response to an input, vertices in case of vertex processors and fragments in case of fragment processors. This is a fundamental difference between C and Cg. Moreover Cg is very specialized for rendering real time graphics and cannot be used to create general-purpose applications as in case of C. This specialized design of the language makes it much faster at graphics computation when compared to C.

3.5 Cg Compiler and Runtime

Cg programs cannot be executed in their original textual form, and needs to be converted to a language understood by the GPU. This transformation takes place through a process called compilation. First, the Cg program is converted into a 3D programming interface called OpenGL. Secondly, this OpenGL code is transformed to a machine language using OpenGL drivers. The mechanics of the compilation depends on the type of graphics processor and the 3D API. *Figure 2* below illustrates this relationship between various entities.

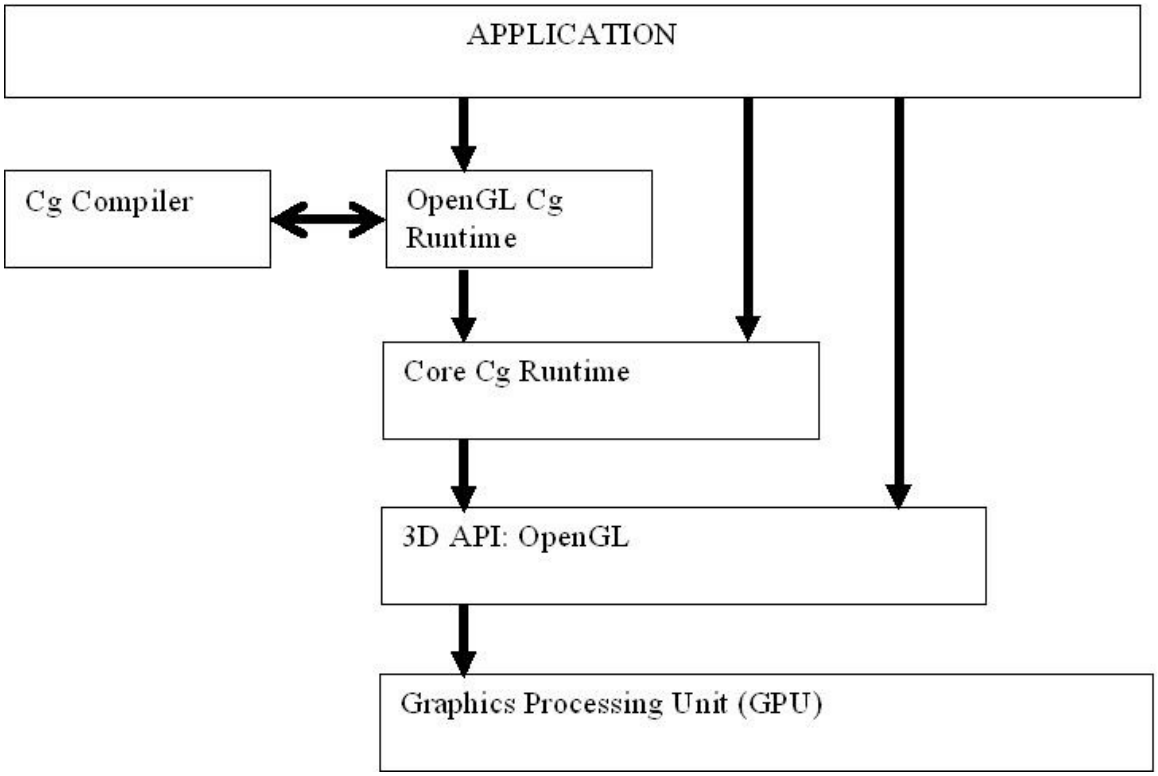


Figure 2. Cg Programming Interface

4 IMPLEMENTATION OF DC POWER FLOW BASED CONTINGENCY ANALYSIS ON GRAPHICS PROCESSING UNIT

4.1 Overview

There are six key features for mapping contingency analysis algorithm onto the GPU, which are mentioned below:

A. Transforming the algorithm into a SIMD algorithm for executing on each of the fragment processor.
B. Choosing appropriate texture parameters.
C. Reshaping all one-dimensional vectors into two dimensional data layout for faster execution.
D. Exploiting the sparsity of the **B** matrix.
E. Setting up indirection matrices.
F. Utilizing all four channels namely: Red, Green, Blue, and Alpha for running four contingencies in parallel.

4.2 Algorithm Mapping

In computing, ***SIMD*** (**S**ingle **I**nstruction, **M**ultiple **D**ata) is a technique employed to achieve data level parallelism, as in a vector or array processor. As the name suggests a single instruction is operated on all the streams of data, unlike traditional CPU programming model where there are multiple instructions and multiple data. This is a

fundamental design difference between GPU and CPU programming model. Transforming the algorithm to SIMD fashion is implemented by writing a "fragment program" which is executed on each of the data elements. In this case our basic computational kernel is**:**

$$\Delta\boldsymbol{\delta}[i+1] = \Delta\boldsymbol{\delta}\left[i\right] - \mathbf{D}^{-1}\mathbf{B}\Delta\boldsymbol{\delta}\left[i\right] + \mathbf{D}^{-1}\Delta\mathbf{P} \tag{14}$$

with

$$\mathbf{D} = \begin{pmatrix} B_{11} & & 0 \\ & \ddots & \\ 0 & & B_{nn} \end{pmatrix} \tag{15}$$

i, is the i^{th} iteration

$\Delta\boldsymbol{\delta}$ is the voltage angle correction texture.

$\Delta\mathbf{P}$ is the real power mismatch texture.

D is the texture comprised of the diagonal elements of the **B** matrix.

We can think of the above kernel as being operated on each of the elements of the $\Delta\boldsymbol{\delta}$ texture, independently. Alternatively, it can be interpreted that each of the element of the $\Delta\boldsymbol{\delta}[i+1]$ texture represents a pixel on the screen. The 24 parallel fragment processors of the NVIDIA 7800 GTX, operate on each of the pixels. From a graphical point of view, the processor is merely trying to illuminate each of the pixels in the target texture $\Delta\boldsymbol{\delta}$[i+1], which is attached to the frame buffer . In order to compute the final color of the pixel as rendered onto the screen it needs to lookup the values stored in the input textures $\Delta\mathbf{P}$, $\Delta\boldsymbol{\delta}$[i+1], **B,** and indirection textures ***B_x*** and ***B_y*** (to be explained in section E).

One aspect, which needs further clarification, is the fact that unlike CPU memory, two-dimensional GPU textures cannot be accessed randomly. At any given point of time during computation, the texture could be read only or write only. This implies that we cannot simply read as well as write to the $\Delta\boldsymbol{\delta}$ texture, as would be the case in a normal CPU programming. In order to overcome this difficulty we store $\Delta\boldsymbol{\delta}$ in a double-buffered texture, alternately serving as a render target and source texture. This technique has been frequently used for general-purpose computation on GPU and is referred to as *"Ping-Pong"* technique [9].

4.3 Choosing Appropriate Texture Parameters

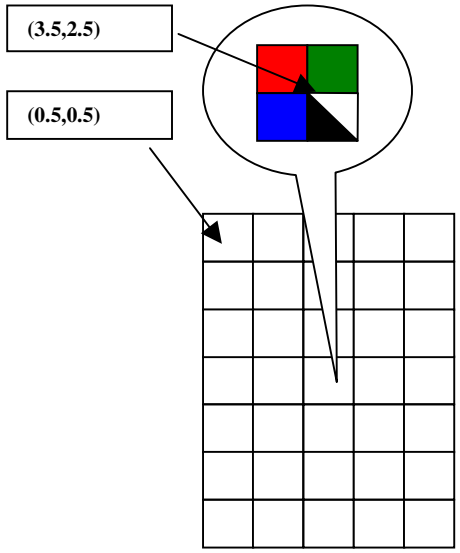


Fig: 3 Co-ordinate System

As shown in *Figure 3* each of the texture co-ordinates refers to a set of four-color components. A single texture fetch operation in graphics computing would return floating point values (color intensity) for each of the *RGBA* colors. In terms of power system computation this translates to a set of four floating-point values storing the elements of the three textures **B,** $\Delta\mathbf{P}$, $\Delta\boldsymbol{\delta}$ and indirection textures**.** The co-ordinates are shifted by a

value of 0.5 as we have chosen Rectangular Textures as our texture target as opposed to texture 2D. The benefits of these are outlined in the *Table 1*.

Table:1 Texture Configurations

	Texture2D	**Texture Rectangle**
Texture Target	GL_TEXTURE_2D	GL_TEXTURE_RECTANGLE_ARB
Texture Coordinates	Coordinates have to be normalized to the range [0,1] by [0,1].	Coordinates are not normalized.
Texture Dimensions	Dimensions are constrained to powers of two (e.g. 1024 by 512)	Dimensions can be arbitrary by definition, e.g. 513 by 1025.

Courtesy: www.gpgpu.org

The next important design choice we made was in selecting the texture **format**. GPUs allow for the simultaneous processing of scalars, tupels, triples or four-tuples of data. To use all four channels, the texture format is **GL_RGBA**. This means that we store four floating-point values per texel, one in the red color channel, one in the green channel and so on. For single precision floating point values, a RGBA texture requires 4*32=128 bits (16 bytes) per texel.

4.4 Reshaping Data from One Dimensional CPU Matrices to Two Dimensional GPU Textures

Graphics processors are traditionally designed to render two-dimensional images onto the screen. Hence the architecture is designed to operate on two-dimensional data structures stored as texture maps. Reshaping the data from one-dimensional CPU layout to two-dimensional GPU layout is critical for faster execution. As shown in *Figure 4*, the uni-dimensional vectors $\Delta\mathbf{P}$ and $\Delta\boldsymbol{\delta}$ of equation (7) could be wrapped around to generate a

two-dimensional data layout. The width and height of the resultant two-dimensional layout depends on the size of the initial vector. In the event of the vector length happens to be a prime number, we pad the initial vector with zeros in order to obtain a two dimensional representation. The **B** matrix is appropriately plugged with zeros in order to reflect the change in $\Delta\mathbf{P}$ and $\Delta\boldsymbol{\delta}$ vectors. As a result of the layout change of the vectors equivalence has to be established in order to relate elements of the **B** matrix to the element of the $\Delta\mathbf{P}$ and $\Delta\boldsymbol{\delta}$ vectors.

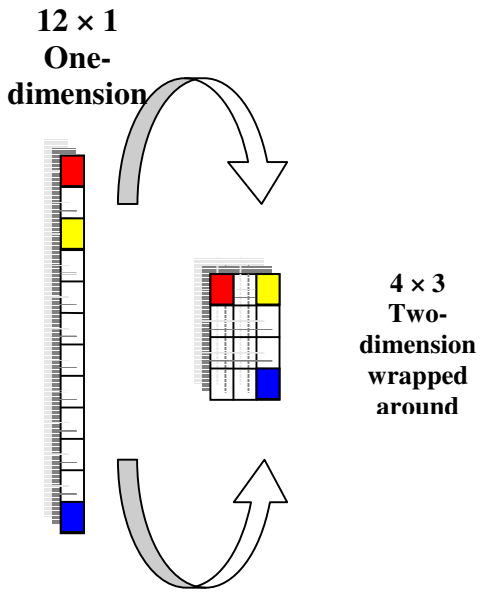


Fig. 4: Conversion of Uni-Dimensional CPU Layout to Two-Dimensional Layout

4.5 Exploiting the Sparsity of the B Matrix

Although the matrix **B** is already in a two dimensional layout, it needs further modification in order to take advantage of the sparsity. We take the following approach to reduce the size of the matrix.

Firstly, the row with maximum number of non-zero entries is detected. In a typical power system this is usually ≈10, irrespective of the system size. This is due to the fact that for any system, typically, the maximum number of busses to which any given bus is connected is ≈10.

Secondly, once the bus with maximum number of non-zero elements is detected, this sets the width of the resulting **B** matrix. The length of the matrix remains the same provided; it has not been compensated with zeros in order to match the $\Delta\mathbf{P}$ and $\Delta\boldsymbol{\delta}$ vectors.
Having established the size of the resulting squeezed matrix, the program initializes the matrix elements to zero. Then it goes through each of the row of the initial matrix serially, and places the non-zero elements, in contiguous locations starting from row 1. This technique leads to a 99% decrease in the size of the matrix for a thousand-bus system, thereby saving a significant number of floating point operations.

4.6 Setting Up the Indirection Matrix

In order to establish equivalence between the three two dimensional matrix elements **B,** $\Delta\mathbf{P}$ and $\Delta\boldsymbol{\delta}$, we need to have two indirection matrices, one storing the X co-ordinates of the matching entry in the wrapped around $\Delta\mathbf{P}$ and $\Delta\boldsymbol{\delta}$ vectors, and the other the respective Y co-ordinate. As we would later see that storing the coordinates (X &Y) in two separate

matrices (textures), helps us execute four contingencies in parallel This indirection matrix has the same structure as the modified **B** matrix, as only non zero entries in the **B** matrix have a corresponding entry in the indirection matrices, the rest of the elements of the indirection matrices are zeros. As shown in Fig. 5, let's assume that the cell colored pink in the **B** texture needs to access the cell (1.5,0.5) in the wrapped around $\Delta\boldsymbol{\delta}$ texture (also marked pink). The X and Y texture co-ordinates of the equivalent cell in the $\Delta\boldsymbol{\delta}(i)$ texture has been stored at the same texture co ordinate location in textures **B**_x (1.5) and **B**_y (0.5). Hence for any element in the **B**-texture, to access its equivalent in the $\Delta\boldsymbol{\delta}(i)$ texture two additional texture lookups are required. Furthermore, no co-ordinate manipulation is required, as the texture co-ordinates of the cell holding the X and Y coordinates in **B**_x and **B**_y respectively, are exactly the same as the texture co ordinates in the **B** matrix.

In addition, once the elements of the $\Delta\boldsymbol{\delta}(i+1)$ texture is computed, the results are already in the right location. Hence we can use the same unwrapping routine without any further modification.

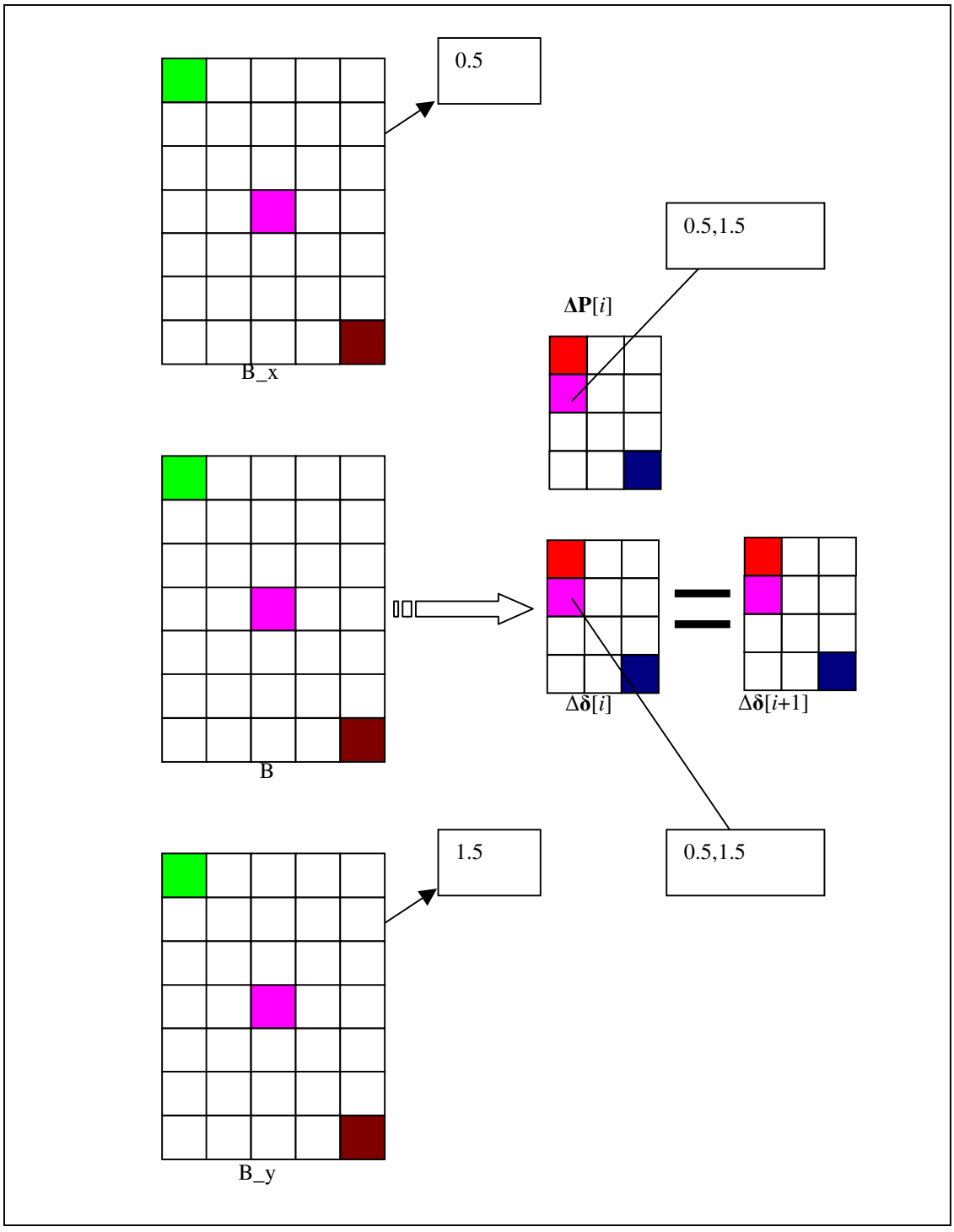


Fig. 5: Data Mapping

4.7 Utilizing All Four RGBA Channels in Parallel

GPU's are designed to operate on four tuples of data concurrently. These data elements are the four color components, *Red, Blue, Green and Transparency (alpha),* required to illuminate each pixel locations. We utilize each of the four channels of the GPU for computation. Doing so, allows us to execute four contingencies in parallel without any additional execution time penalty. As each of the contingencies computations is inherently parallel, blocks of four contingencies are presented for the GPU in each of the channels, thereby reducing the execution time. Since the structure of the **B** matrix, once computed, remains constant for all the contingencies, this allows executing blocks of four contingencies in parallel. In an *SIMD* programming model a single instruction is executed to compute each element of the $\Delta\boldsymbol{\delta}$ vector. It is thus required to pre-compute the indirection textures, establishing the mapping between the **B**, $\boldsymbol{\Delta P}$ and $\Delta\boldsymbol{\delta}$**,** textures. Having computed the indirection textures once, it is not necessary to reload the texture for each set of four contingencies along with **B** textures. The **B** texture would be different for each set of contingencies depending upon the outage case considered.

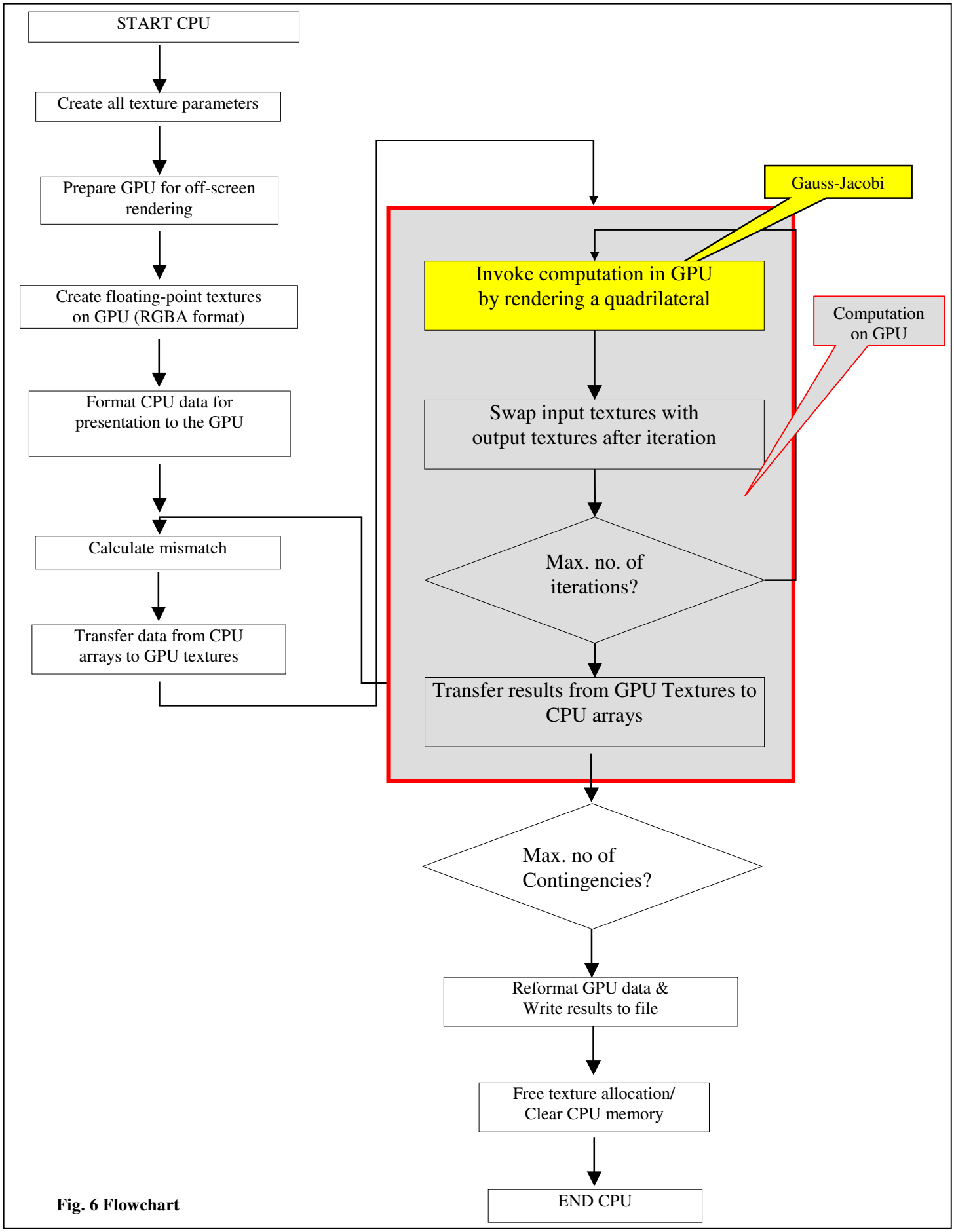


Fig. 6 Flowchart

5 SAMPLE IMPLEMENTATION

5.1 Overview

Based on our discussion in the previous chapter, we are going to introduce a sample four bus system, and go through the six key features for mapping contingency analysis algorithm onto the GPU, which are mentioned below.

1. Transforming the algorithm into a SIMD algorithm for executing on each of the fragment processor.
2. Choosing appropriate texture parameters.
3. Reshaping all one-dimensional vectors into two dimensional data layout for faster execution.
4. Exploiting the sparsity of the **B** matrix.
5. Setting up indirection matrices.
6. Utilizing all four channels namely: Red, Green, Blue, and Alpha for running four contingencies in parallel.

Figure 7 shows the one-line diagram of the sample system. Generators are connected at busses 1 and 4 while loads are indicated at all four busses. The base values for transmission are 100MVA and 230KV. The line data of *table 1* give per unit series impedances and line charging susceptances for the nominal-π equivalent of the four lines identified by the busses at which they terminate. The bus data in *table 3* list active and reactive power values P_i and Q_i and the voltage phasor values V_i at each bus, i=1, …,4. The Q_i are calculated from the corresponding P_i assuming a power factor of 0.85.

5.2 The Sample System

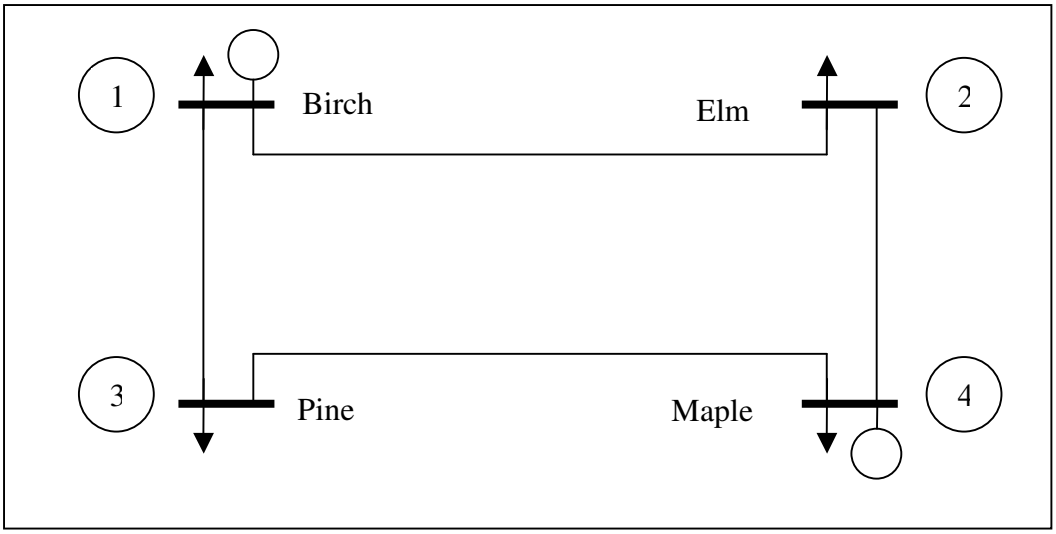


Fig. 7 One Line Diagram

Table 2: Line Data

Line	*R Perunit*	*X Perunit*	*G Perunit*	*B Per unit*	*charging Mvar*	*Y/2Per unit*
1-2	0.01008	0.05040	3.815629	-19.078144	10.25	0.05125
1-3	0.00744	0.03720	5.169561	-25.847809	7.75	0.03875
2-4	0.00744	0.03720	5.169561	-25.847809	7.75	0.03875
3-4	0.01272	0.06360	3.023705	-15.118528	12.75	0.06375

Table 3: Bus Data

	Generation		*Load*			
Bus	**P,MW**	**Q,Mvar**	**P,MW**	**Q,Mvar**	**V,per unit**	**Remarks**
1	-	-	50	30.99	$1.00\angle 0^0$	Slack Bus
2	0	0	170	105.35	$1.00\angle 0^0$	Load Bus
3	0	0	200	123.94	$1.00\angle 0^0$	Load Bus
4	318	-	80	49.58	$1.00\angle 0^0$	Voltage Controlled

Table 4: Y-Bus Admittance Matrix

Bus No.	***1***	***2***	***3***	***4***
1	8.985190-j44.835953	-3.815629+j19.078144	-5.169561+j25.847809	0
2	-3.815629+j19.078144	8.985190-j44.835953	0	-5.169561+j25.847809
3	-5.169561+j25.847809	0	8.193267-j40.863838	-3.023705+j15.118528
4	0	-5.169561+j25.847809	-3.023705+j15.118528	8.193267-j40.863838

5.3 Problem Formulation

For the above system, based on our discussions in chapter 2 the resulting DC power flow model for the system would be:

$$\mathbf{B}\Delta\boldsymbol{\delta} = \Delta\mathbf{P} \tag{16}$$

$$\begin{pmatrix} 44.835 & 0 & -25.847 \\ 0 & 40.863 & -15.1185 \\ -25.847 & -15.118 & 40.8638 \end{pmatrix} \begin{pmatrix} \Delta\delta_2 \\ \Delta\delta_3 \\ \Delta\delta_4 \end{pmatrix} = \begin{pmatrix} -1.5966 \\ -1.939 \\ 2.210 \end{pmatrix} \tag{17}$$

Here

B = *Imag*(**Y**) **denotes** the bus reactance matrix (excluding the slack bus

$\Delta\boldsymbol{\delta}$ presents the angle mismatch vector to be computed.

$\Delta\mathbf{P}$ presents the real power mismatch vector at each bus except the slack bus.

5.4 Data Manipulation

First, before performing any computation in the GPU, the data has to be manipulated in the CPU based on our discussions in the previous chapter. A step by step approach is presented for reformulating the data for GPU computation

5.4.1 Step 1. Exploiting Sparsity

As we can see the **B** matrix in this case is non sparse and hence we cannot apply the sparsity technique discussed in the previous chapter. However this is not usually the case in power system computation.

5.4.2 Step 2. Padding the Matrix for Conversion to Two Dimensional Format

Since the $\Delta\boldsymbol{\delta}$ and $\Delta\mathbf{P}$ vectors have 3 elements each, we pad them with one zero and correspondingly add a row and a column of zeros to the **B** matrix. This is done to achieve a two dimensional representation for the vectors. This step is optional and has to be performed only when the number of elements in the $\Delta\boldsymbol{\delta}$ and $\Delta\mathbf{P}$ vectors are prime numbers.

For example if the number of elements in the vector is 16 it can be easily broken down in (4×4) or if its 15 it can be broken down into (3×5); on the other hand if the length of the vector is 17, it cannot be split and hence by padding it with one additional zero would bring the length to 18 and can be split into a matrix with dimension (3×6).

After performing the above mentioned manipulation. The resulting matrices would be:

$$\mathbf{B}=\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 44.835 & 0 & -25.847 \\ 0 & 0 & 40.863 & -15.1185 \\ 0 & -25.847 & -15.118 & 40.8638 \end{pmatrix}$$

$$\Delta\boldsymbol{\delta}=\begin{pmatrix} 0 \\ \Delta\delta_2 \\ \Delta\delta_3 \\ \Delta\delta_4 \\ \end{pmatrix}$$

$$\Delta\mathbf{P} = \begin{pmatrix} 0 \\ -1.5966 \\ -1.939 \\ 2.210 \\ \end{pmatrix}$$

5.4.3 Step 3. Converting from One Dimensional CPU Layout To Two Dimensional GPU Layout

Once the vectors and matrices are padded we are ready to convert them to two-dimensional representation. These two dimensional matrices would then be mapped to two-dimensional textures in GPU memory. Since the **B** matrix is already two-dimensional it does not need any further modification. We would wrap around the two vectors $\Delta\boldsymbol{\delta}$ and $\Delta\mathbf{P}$, which are shown below.

$$\Delta\boldsymbol{\delta} = \begin{pmatrix} 0 & \Delta\delta_2 \\ \Delta\delta_3 & \Delta\delta_4 \end{pmatrix} \quad \text{and} \quad \Delta\mathbf{P} = \begin{pmatrix} 0 & -1.5966 \\ -1.939 & 2.210 \end{pmatrix}$$

5.4.4 Step 4. Computing Indirection Matrices

Once the dimensions of the two vectors are modified from one to two-dimensional form, the spatial correspondence between the elements of the **B** matrix and the $\Delta\boldsymbol{\delta}$ and $\Delta\mathbf{P}$ vectors are lost. In order to store this information we need to have two indirection matrices one specifying the X-coordinate and the other Y-coordinates. These matrices establish the relationship between each element of **B** matrix and the corresponding element in the vectors. Indirection matrices assume increased significance when sparsity techniques are used, where the **B** matrix is reshaped. In this example this is not the case.

These matrices are again stored in the form of textures in the GPU memory. Additionally while performing computation on each stream of data in the fragment processor, these matrices are accessed to define the relationship.

As we discussed in the previous chapter, the coordinate system in the GPU memory is offset by 0.5, as it refers to Texel centers, moreover the origin of the coordinates is at the top left corner of the screen. This is in contrast with the convention, where the coordinate system is at the bottom left corner.

One additional detail needs to be explained. In the case where the matrices are padded with zeros the indirection matrices hold the value of 0.5 (both X and Y) in the padded rows and columns, this is because the location (0.5,0.5) refers to the first element in the reshaped vectors, which is zero. As these padded number do not take part in the calculation, and are there merely to get a two dimensional layout, the result of the calculation is also zero.

Based on the discussions, the indirection matrices **B**_x and **B**_y are shown below.

$$\mathbf{B}\_x = \begin{pmatrix} 0.5 & 0.5 & 0.5 & 0.5 \\ 0.5 & 1.5 & 0.5 & 1.5 \\ 0.5 & 1.5 & 0.5 & 1.5 \\ 0.5 & 1.5 & 0.5 & 1.5 \end{pmatrix} \quad \mathbf{B}\_y = \begin{pmatrix} 0.5 & 0.5 & 0.5 & 0.5 \\ 0.5 & 0.5 & 1.5 & 1.5 \\ 0.5 & 0.5 & 1.5 & 1.5 \\ 0.5 & 0.5 & 1.5 & 1.5 \end{pmatrix}$$

It is important to note that the matrices will be of exact same dimensions as the **B** matrix and there will be a one to one correspondence between element of the two indirection matrices and the elements of the **B** matrix.

5.5 Uploading Matrices to GPU Memory

This completes are initial data preparation stage for one contingency. We need to repeat the steps above for each of the contingency. As the system topology changes with each contingency, the **B** matrix and $\Delta\mathbf{P}$ vector needs to be reformulated, for each contingency. However the indirection matrix does not need to be recomputed and does not need to be reloaded with each contingency. Since the computation is performed parallel in four different channels, we need to upload four sets of **B** matrices along with their $\Delta\mathbf{P}$ vector.

There is a scope of improvement in this methodology and can be optimized further by not uploading the entire **B** matrix each time for new contingency. We could rather, upload a base case for once, and then make incremental changes to the existing base case, for each contingency. This would save us time, in communication between GPU and CPU and reduce the traffic on the PCI-express bus. However we have not yet implemented this technique, but would be interesting to see the saving in time due to this approach.

5.6 Attaching Matrices to Textures

On uploading the matrices to the GPU memory, we attach them to the textures which are internal storage for GPU. Each matrix corresponding to a particular contingency is attached to one channel of the 4 dimensional textures. The size of the textures are predefined based on the size of each two-dimensional matrix.

5.7 Loading Fragment Processors

The fragment processors are loaded with a fragment program, which basically operates on each data stream input to the processor once the computation is invoked. In our case the fragment program is the:

$$\Delta\boldsymbol{\delta}[i+1] = \Delta\boldsymbol{\delta}\left[i\right] - \mathbf{D}^{-1}\mathbf{B}\Delta\boldsymbol{\delta}\left[i\right] + \mathbf{D}^{-1}\Delta\mathbf{P} \tag{18}$$

where,

$$\mathbf{D} = \begin{pmatrix} B_{11} & & 0 \\ & \ddots & \\ 0 & & B_{nn} \end{pmatrix}$$

i, is the ith iteration.

$\Delta\boldsymbol{\delta}$ is the voltage angle correction texture.

$\Delta\mathbf{P}$ is the real power mismatch texture.

D is the texture comprised of the diagonal elements of the **B** matrix.

The Source code modeling this fragment processor program is provided in the *Appendix C.*

5.8 Starting Computation

Once the fragment processor is loaded with the fragment program and the textures are populated with four sets of contingencies. The computation is invoked by asking the GPU to render a quadrilateral. This sets of the computation for a predefined number of iterations.

5.9 Downloading Results.

Since in our case the system comprises of just four busses, there is no need to load another set of contingencies and all the four contingencies would be solved in one shot. Hence, we can download the results from the GPU textures, as the final result of computation.

5.10 Garbage Collection

Once the results are imported the CPU memory is cleared and the GPU textures are destroyed. This completes the simulation.

6 SIMULATION RESULTS

6.1 Overview

The following tables 5-8 show the simulation results for sparse matrix computation using a sparse-matrix Gauss-Jacobi/Gauss-Siedel algorithm implemented on the CPU and the GPU. Fig. 7 shows a relative comparison between the speedup as the system size changes. Fig 8 show a comparison between the two algorithms on CPU and GPU. We have used a single core Pentium 4, 3.2 GHz , 2GB R.A.M & NVIDIA 7800 GTX card with 256 MB Memory, PCI Express graphics bus technology and memory bandwidth of 54.4 GB/sec . While the IEEE 118 bus system and the IEEE 300 bus system are standard test systems, we have defined the **B** matrix for a fictitious 1000/2000 bus system based on assumptions of network sparsity, realistic serial and shunt admittance size and Y-bus symmetries. As the results show, the gain in speed of a factor of 4 seems to indicate that the gain is mainly due to the parallelization of the algorithm on four “color” textures. Fig 8 shows a comparison of the Performance analysis with the growth in system size. Regardless of the number of iterations in all the cases there is an appreciable speedup with the increase in system size. This increase in speedup with increase in system size is not infinite, and there would be no additional speedup once the pipeline saturates. In this thesis we have not attempted to determine that limit.

6.2 Results

Table 5: Simulation Results for 118 Bus System

Execution time [s]	10 iterations	100 iterations	1000 iterations
CPU	0.639	1.123	5.923
GPU	0.202	0.483	3.226
Speedup	3.16	2.32	3.97

Table 6: Simulation Results for300 Bus System

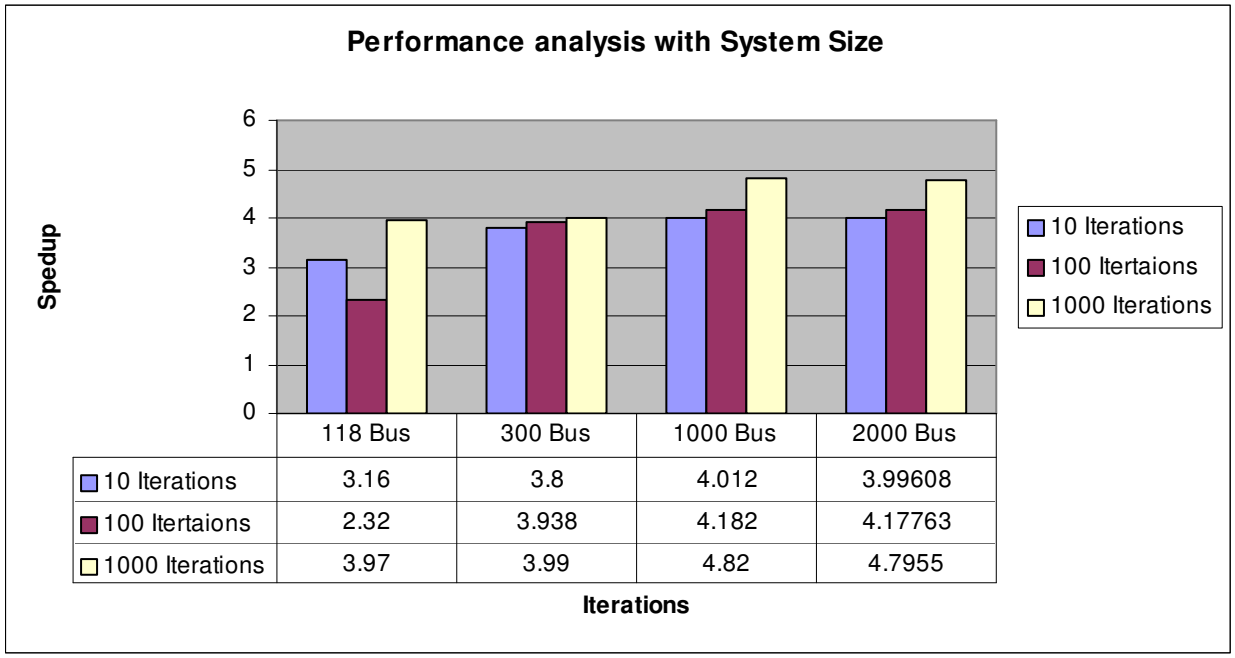
Execution time [s]	10 iterations	100 iterations	1000 iterations
CPU	8.728	11.846	44.062
GPU	2.291	3.008	10.115
Speedup	3.80	3.938	3.99

Table 7: Simulation Results for 1000 Bus System

Execution time [s]	10 iterations	100 iterations	1000 iterations
CPU	352.01	390.15	715.114
GPU	87.718	93.282	148.152
Speedup	4.012	4.182	4.82

Table 8: Simulation Results for 2000 Bus System

Execution time [s]	10 iterations	100 iterations	1000 iterations
CPU	61399.5	65603.9	92400.3
GPU	15362.9	15703.6	19268.5
Speedup	3.99608	4.17763	4.7955



	118 Bus	300 Bus	1000 Bus	2000 Bus
10 Iterations	3.16	3.8	4.012	3.99608
100 Itertaions	2.32	3.938	4.182	4.17763
1000 Iterations	3.97	3.99	4.82	4.7955

Fig. 8 Performance Analysis with System Size

7 CONCLUSION AND FUTURE WORK

7.1 Conclusion

We have successfully implemented the (*N*-1) contingency algorithm on the GPU for the DC power flow. The power flow equations are solved using Gauss-Jacobi iterations. We have shown that compared to a similar implementation on the CPU, there is considerable speedup on the GPU. This improved speed is contributed to the highly pipelined parallel architecture of the GPU's as compared to the serial architecture of the CPU. The fact that these processors are already present in almost all desktop based personal computers implies that this additional speedup comes at no additional cost. The GPU architecture, which has been traditionally designed and highly optimized for displaying graphics, can be effectively used as a co-processor to Pentium for the process of conducting contingency analysis of large scale power systems.

7.2 Future Work

Future work will focus on the optimization of the implementation of the algorithm on the GPU. Tasks to be improved include optimizing the data layout. Advanced iterative algorithms like waveform relaxation algorithms, Newton Raphson, fast decoupled power flow etc. can also be implemented. Finally transient stability can be modeled by a set of differential-algebraic equations. Solving these stiff set of DAEs requires an implicit integration algorithm, which in turn requires Newton-Raphson type iterations. Thus a solid understanding of the GPUs processing paradigms and processing power will address a large class of power system analysis and simulation problems. Recently,

NVIDIA has exposed many aspects of the GPU internals via the CUDA interface. This interface allows us to program the GPU as a general highly parallel stream processor, rather than having to use OpenGL and graphics semantics. It is likely that this extended interface can be exploited to generate increased performance enhancements for linear system solvers.

8 LIST OF REFERENCES

[1] V. C. Ramesh, "On distributed computing for on-line power system applications," *International Journal of Electrical Power & Energy Systems,* vol. 18, pp. 527-533, Nov 1996.

[2] N. Balu, T. Bertram, A. Bose, V. Brandwajn, G. Cauley, D. Curtice, A. Fouad, L. Fink, M. G. Lauby, B. F. Wollenberg, and J. N. Wrubel, "Online Power-System Security Analysis," *Proceedings of the IEEE,* vol. 80, pp. 262-280, Feb 1992.

[3] N. Balu, T. Bertram, A. Bose, V. Brandwajn, G. Cauley, D. Curtice, A. Fouad, L. Fink, M. G. Lauby, B. F. Wollenberg, and J. N. Wrubel, "On-line power system security analysis," *Proceedings of the IEEE,* vol. 80, pp. 262-282, 1992.

[4] M. S. J.Duncan Glover, *Power System Analysis & Design*. Boston: PWS Publishing Company, 1994.

[5] M. H. Naga Govindaraju, Jens Krüger, Aaron E., Lefohn, and Timothy J. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," *Eurographics 2005,* vol. State of the Art Reports, pp. 21-51, August 2005.

[6] M. Pharr, "GPU Gems 2," in *Programming Techniques for High-Performance Graphics and General-Purpose Computation*, R. Fernando, Ed.: Addison-Wesley, 2005, pp. 451-519.

[7] J. G. R. Kuffel, T. Maguire, R. P. Wierckx, and P. G. McLaren, "A fully digital power system simulator operating in real time," *Proc. of Conference on Canadian Electrical and Computer Engineering,* vol. 2, pp. 733-736, 26-29, May 1996.

[8] V.C.Ramesh, "On distributed computing for on-line power system applications," *International Journal of Electrical Power & Energy Systems*, vol. 18, pp. 527-533, 1996.

[9] S. Venkatsubramanian, "The Graphics card as a stream computer," *SIGMOD-DIMACS Workshop on Management and Processing of Data Streams,* 2003.

[10] A. Bose, "*Parallel-Processing in Dynamic Simulation of Power-Systems,*" Sadhana-Academy Proceedings in Engineering Sciences, 1993.

[11] Luo, P, Yang, F. H, Rao, M. "A parallel approach to computing load flow equations," *Dynamics of Continuous Discrete and Impulsive Systems-Series B-Applications & Algorithms,* 2004.

[12] Taoka, H.Iyoda, I.Noguchi, H.Sato, N.,Nakazawa, T. Tinney, W. F.,Chai, J.S., Bose, A. J., *"*Real-Time Digital Simulator for Power-System Analysis on a Hypercube Computer*," IEEE Transactions on Power Systems*, vol. 1,pp. 1-10, 1992.

[13] Bialek, J., Grey, D. J., "Application of Clustering and Factorization Tree Techniques for Parallel Solution of Sparse Network Equations," *IEE Proc.-Gener. Transm. Distrib*., vol. 141, pp. 609, issue 6, 1994.

[14] Gopal, A., Niebur, D., Venkatsubramanian, S., "DC Power Flow Based Contingency Analysis Using Graphics Processing Units," *Proceedings of Power Tech 2007*, Lausanne, Switzerland, July 1-5, 2007.

[15] M. La Scala ***G.*** Sblendorio A. Bose J.Q**.** Wu., "Comparison of Algorithms for Transient Stability Simulations on Shared and Distributed Memory Multiprocessor," *IEEE Transactions on Power Systems*, Vol. 11, No. 4, November 1996.

9 APPENDIX A: Source Code

```
#ifdef __APPLE_CC__
#include <CoreFoundation/CoreFoundation.h>
#endif

// Include the library specific header file as generated by the
// MATLAB Compiler
#include "gpudclib.h"
#include "mclcppclass.h"
#include <iostream>
#include <GL/glew.h>
#include <GL/glut.h>
#include <Cg/cgGL.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <string>
#include "toolbox.h"
#include "shaders.h"
#include "FeastLink.h"
#include "main.h"
#define pi 3.142857143

// problem size, texture size etc
// set by command line, see main(int, char**)
const int N=300;
const int N_actual=300;
const int texWidth_X=10;
const int texHeight_X=30;
const int texWidth_Ybus=11;
const int texHeight_Ybus=300;
const int numIterations=1000;
// CPU
const int NumberOfCPUContingencies=411;
//GPU
const int NumberOfGPUContingencies=103;
const int NumberOfPFIterations =1;
const bool showResults = false;
const float EPSILON = 0.0000005;


// actual data

rgba* dataUpdate=(rgba*)malloc(N*sizeof(rgba));
rgba* dataMiss=(rgba*)malloc(N*sizeof(rgba));
rgba*
dataYbus=(rgba*)malloc(texHeight_Ybus*texWidth_Ybus*sizeof(rgba));
rgba* dataYbus_imag_dd=(rgba*)malloc(N*sizeof(rgba));
rgba* dataYbus_imag_ddinv=(rgba*)malloc(N*sizeof(rgba));
```

```
rgba*
dataI_imag=(rgba*)malloc(texHeight_Ybus*texWidth_Ybus*sizeof(rgba));
rgba*
dataJ_imag=(rgba*)malloc(texHeight_Ybus*texWidth_Ybus*sizeof(rgba));
```

```
// ping pong macros and vars
GLenumattachmentpoints[]=
{GL_COLOR_ATTACHMENT0_EXT,GL_COLOR_ATTACHMENT1_EXT};
int SOURCE_BUFFER = 0;
int DESTINATION_BUFFER =1;


// multitex vars
GLuint fb;
// for  mismatch calculation
GLuint xTexID[2];
GLuint bTexID;
// common textures
GLuint Ybus_imagTexID;
GLuint Ybus_imag_ddinvTexID;
GLuint Ybus_imag_ddTexID;
GLuint I_imagTexID,J_imagTexID;


// Cg vars
CGcontext cgContext;
CGprofile vertexProfile;
CGprofile fragmentProfile;
// vertex programs
CGprogram vertexProgram1;
// handles to vertex program parameters
CGparameter matrixParam;
// handles to access texture samplers in the shader 1: all  texels
CGprogram fragmentProgram1;
CGparameter xParam1, bParam1;
CGparameter Ybus_imagParam1, Ybus_imag_ddinvParam1;
CGparameter I_imagParam1,J_imagParam1, Ybus_imag_ddParam1;
```

```
// Logger
Logger *logger;

// window handle
GLuint windowHandle;
struct struct_textureParameters {
      char* name;
```

```
      GLenum texTarget;
      GLenum texInternalFormat;
      GLenum texFormat;
      }     rect_arb_rgba_32, // texture rectangles, texture_float_ARB,
RGBA, 32 bits
      rect_arb_rgba_16, // texture rectangles, texture_float_ARB, RGBA,
16 bits
rect_arb_r_32,          // texture rectangles, texture_float_ARB, R, 32
bits
rect_ati_rgba_32, //  texture  rectangles,  ATI_texture_float,  RGBA,  32
bits
rect_ati_rgba_16, //  texture  rectangles,  ATI_texture_float,  RGBA,  16
bits
rect_ati_r_32,          // texture rectangles, ATI_texture_float, R, 32
bits
rect_nv_rgba_32,  // texture rectangles, NV_float_buffer, RGBA, 32 bits
rect_nv_rgba_16,  // texture rectangles, NV_float_buffer, RGBA, 16 bits
rect_nv_r_32,           //  texture  rectangles,  NV_float_buffer,  R,  32
bits
rect_nv_arb_r_32,      //texture  rectangles,  NV_float_buffer,  R,  32
bits(from jacobi)
twod_arb_rgba_32, // texture 2ds, texture_float_ARB, RGBA, 32 bits
twod_arb_rgba_16, // texture 2ds, texture_float_ARB, RGBA, 16 bits
twod_arb_r_32,          // texture 2ds, texture_float_ARB, R, 32 bits
twod_ati_rgba_32, // texture 2ds, ATI_texture_float, RGBA, 32 bits
twod_ati_rgba_16, // texture 2ds, ATI_texture_float, RGBA, 16 bits
twod_ati_r_32,          // texture 2ds, ATI_texture_float, R, 32 bits
twod_nv_rgba_32,  // texture 2ds, NV_float_buffer, RGBA, 32 bits
twod_nv_rgba_16,  // texture 2ds, NV_float_buffer, RGBA, 16 bits
twod_nv_r_32;           // texture 2ds, NV_float_buffer, R, 32 bits

// struct actually being used (set from command line)
struct_textureParameters textureParameters;



// timing vars
LARGE_INTEGER ticksPerSecond;
LARGE_INTEGER start_ticks, end_ticks, total_time;
// timing vars
double start, end;
double total_upload=0.0000;
double total_init=0.0000;
double total_comp=0.0000;
double total_download=0.0000;
double total_comp_CPU=0.0000;




// forward declarations
//void displayCallback(void);
void printInfo(void);
```

```
/*
 * Callback for Cg errors
 */
void cgErrorCallback() {
      CGerror lastError = cgGetError();
      if(lastError) {
            logger->fatal(cgGetErrorString(lastError));
            logger->fatal(cgGetLastListing(cgContext));
            exit(ERROR_CG);
      }
}


void createAllTextureParameters(void) {
rect_arb_rgba_32.name= "TEXRECT - float_ARB - RGBA - 32";
rect_arb_rgba_32.texTarget= GL_TEXTURE_RECTANGLE_ARB;
rect_arb_rgba_32.texInternalFormat= GL_RGBA32F_ARB;
rect_arb_rgba_32.texFormat= GL_RGBA;

rect_arb_rgba_16.name   = "TEXRECT - float_ARB - RGBA - 16";
rect_arb_rgba_16.texTarget= GL_TEXTURE_RECTANGLE_ARB;
rect_arb_rgba_16.texInternalFormat= GL_RGBA16F_ARB;
rect_arb_rgba_16.texFormat= GL_RGBA;

rect_arb_r_32.name= "TEXRECT - float_ARB - R - 32";
rect_arb_r_32.texTarget= GL_TEXTURE_RECTANGLE_ARB;
rect_arb_r_32.texInternalFormat           = GL_LUMINANCE32F_ARB;
rect_arb_r_32.texFormat                   = GL_LUMINANCE;

rect_ati_rgba_32.name= "TEXRECT - float_ATI - RGBA - 32";
rect_ati_rgba_32.texTarget= GL_TEXTURE_RECTANGLE_ARB;
rect_ati_rgba_32.texInternalFormat= GL_RGBA_FLOAT32_ATI;
rect_ati_rgba_32.texFormat= GL_RGBA;

rect_ati_rgba_16.name= "TEXRECT - float_ATI - RGBA - 16";
rect_ati_rgba_16.texTarget= GL_TEXTURE_RECTANGLE_ARB;
rect_ati_rgba_16.texInternalFormat= GL_RGBA_FLOAT16_ATI;
rect_ati_rgba_16.texFormat= GL_RGBA;

rect_ati_r_32.name= "TEXRECT - float_ATI - R - 32";
rect_ati_r_32.texTarget = GL_TEXTURE_RECTANGLE_ARB;
rect_ati_r_32.texInternalFormat= GL_LUMINANCE_FLOAT32_ATI;
rect_ati_r_32.texFormat= GL_LUMINANCE;

rect_nv_rgba_32.name= "TEXRECT - float_NV - RGBA - 32";
rect_nv_rgba_32.texTarget= GL_TEXTURE_RECTANGLE_ARB;
rect_nv_rgba_32.texInternalFormat= GL_FLOAT_RGBA32_NV;
rect_nv_rgba_32.texFormat= GL_RGBA;
```

```
rect_nv_rgba_16.name= "TEXRECT - float_NV - RGBA - 16";
rect_nv_rgba_16.texTarget= GL_TEXTURE_RECTANGLE_ARB;
rect_nv_rgba_16.texInternalFormat= GL_FLOAT_RGBA16_NV;
rect_nv_rgba_16.texFormat= GL_RGBA;

rect_nv_r_32.name = "TEXRECT - float_NV - R - 32";
rect_nv_r_32.texTarget= GL_TEXTURE_RECTANGLE_ARB;
rect_nv_r_32.texInternalFormat= GL_FLOAT_R32_NV;
rect_nv_r_32.texFormat= GL_LUMINANCE;

rect_nv_arb_r_32.name= "TEXRECT - float_NV_ARB - R - 32";
rect_nv_arb_r_32.texTarget= GL_TEXTURE_RECTANGLE_NV;
rect_nv_arb_r_32.texInternalFormat= GL_FLOAT_R32_NV;
rect_nv_arb_r_32.texFormat= GL_RED;


//////////////

twod_arb_rgba_32.name= "tex2D - float_ARB - RGBA - 32";
twod_arb_rgba_32.texTarget= GL_TEXTURE_2D;
twod_arb_rgba_32.texInternalFormat= GL_RGBA32F_ARB;
twod_arb_rgba_32.texFormat= GL_RGBA;

twod_arb_rgba_16.name= "tex2D - float_ARB - RGBA - 16";
twod_arb_rgba_16.texTarget                = GL_TEXTURE_2D;
twod_arb_rgba_16.texInternalFormat  = GL_RGBA16F_ARB;
twod_arb_rgba_16.texFormat                = GL_RGBA;

twod_arb_r_32.name= "tex2D - float_ARB - R - 32";
twod_arb_r_32.texTarget = GL_TEXTURE_2D;
twod_arb_r_32.texInternalFormat= GL_LUMINANCE32F_ARB;
twod_arb_r_32.texFormat= GL_LUMINANCE;

twod_ati_rgba_32.name= "tex2D - float_ATI - RGBA - 32";
twod_ati_rgba_32.texTarget= GL_TEXTURE_2D;
twod_ati_rgba_32.texInternalFormat= GL_RGBA_FLOAT32_ATI;
twod_ati_rgba_32.texFormat= GL_RGBA;

twod_ati_rgba_16.name= "tex2D - float_ATI - RGBA - 16";
twod_ati_rgba_16.texTarget= GL_TEXTURE_2D;
twod_ati_rgba_16.texInternalFormat= GL_RGBA_FLOAT16_ATI;
twod_ati_rgba_16.texFormat= GL_RGBA;

twod_ati_r_32.name= "tex2D - float_ATI - R - 32";
twod_ati_r_32.texTarget= GL_TEXTURE_2D;
twod_ati_r_32.texInternalFormat= GL_LUMINANCE_FLOAT32_ATI;
twod_ati_r_32.texFormat = GL_LUMINANCE;

twod_nv_rgba_32.name= "tex2D - float_NV - RGBA - 32";
twod_nv_rgba_32.texTarget= GL_TEXTURE_2D;
twod_nv_rgba_32.texInternalFormat= GL_FLOAT_RGBA32_NV;
twod_nv_rgba_32.texFormat= GL_RGBA;

twod_nv_rgba_16.name= "tex2D - float_NV - RGBA - 16";
twod_nv_rgba_16.texTarget= GL_TEXTURE_2D;
twod_nv_rgba_16.texInternalFormat= GL_FLOAT_RGBA16_NV;
twod_nv_rgba_16.texFormat= GL_RGBA;
```

```
twod_nv_r_32.name= "tex2D - float_NV - R - 32";
twod_nv_r_32.texTarget= GL_TEXTURE_2D;
twod_nv_r_32.texInternalFormat= GL_FLOAT_R32_NV;
twod_nv_r_32.texFormat= GL_LUMINANCE;

}
/*
 * utility function to set up texture params
 */
void setupTexture (const int texARB,const GLuint texID,int t_w,int t_h)
{
      int texWidth=t_w;
      int texHeight=t_h;
      glActiveTextureARB(texARB);
      // make active and bind
      glBindTexture(textureParameters.texTarget,texID);
      // set parameters that suit our viewport
      glTexParameteri(textureParameters.texTarget,
GL_TEXTURE_MIN_FILTER, GL_NEAREST);
      glTexParameteri(textureParameters.texTarget,
GL_TEXTURE_MAG_FILTER, GL_NEAREST);
      glTexParameteri(textureParameters.texTarget,    GL_TEXTURE_WRAP_S,
GL_CLAMP);
      glTexParameteri(textureParameters.texTarget,    GL_TEXTURE_WRAP_T,
GL_CLAMP);

      glTexImage2D(textureParameters.texTarget,0,textureParameters.texI
nternalFormat,texWidth,texHeight,0,textureParameters.texFormat,GL_FLOAT
,0);
}

void transferToTexture (rgba* data, GLuint texID,int t_w,int t_h) {

      int texWidth=t_w;
      int texHeight=t_h;
      glBindTexture(textureParameters.texTarget, texID);
      glTexSubImage2D(textureParameters.texTarget,0,0,0,texWidth,texHei
ght,textureParameters.texFormat,GL_FLOAT,data);


}
/*
 * Downloads data from given target (GL_FRONT_LEFT | GL_BACK_LEFT).
 */
void transferFromTexture(rgba* data,int t_w,int t_h)
{
      int texWidth=t_w;
      int texHeight=t_h;
  glReadBuffer(attachmentpoints[SOURCE_BUFFER]);
      glReadPixels(0,                    0,                    texWidth,
texHeight,textureParameters.texFormat,GL_FLOAT,data);

}


/*
```

```
 * Creates offscreen render surface, configures it and creates all
neccessary textures.
 */
void createTextures(void)
{

      glEnable(textureParameters.texTarget);
      // create texture ids and targets
      glGenTextures (2, xTexID);
      glGenTextures (1, &bTexID);
      glGenTextures (1, &Ybus_imagTexID);
      glGenTextures (1, &Ybus_imag_ddinvTexID);
      glGenTextures (1, &I_imagTexID);
      glGenTextures (1, &J_imagTexID);
      glGenTextures (1, &Ybus_imag_ddTexID);




      setupTexture
(GL_TEXTURE0_ARB,xTexID[SOURCE_BUFFER],texWidth_X,texHeight_X);
      setupTexture
(GL_TEXTURE1_ARB,xTexID[DESTINATION_BUFFER],texWidth_X,texHeight_X);
      setupTexture (GL_TEXTURE2_ARB,bTexID,texWidth_X,texHeight_X);
      setupTexture
(GL_TEXTURE3_ARB,Ybus_imagTexID,texWidth_Ybus,texHeight_Ybus);
      setupTexture
(GL_TEXTURE4_ARB,Ybus_imag_ddinvTexID,texWidth_X,texHeight_X);
      setupTexture
(GL_TEXTURE5_ARB,I_imagTexID,texWidth_Ybus,texHeight_Ybus);
      setupTexture
(GL_TEXTURE6_ARB,J_imagTexID,texWidth_Ybus,texHeight_Ybus);
      setupTexture
(GL_TEXTURE7_ARB,Ybus_imag_ddTexID,texWidth_X,texHeight_X);




}

/*
 * Inits for Cg runtime, shader creation, shader parameter handle
aquisition.
 */
void initFBO(void) {
            // create FBO (off-screen framebuffer)
  glGenFramebuffersEXT(1, &fb);
  // bind offscreen framebuffer (that is, skip the window-specific
render target)
  glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
  // viewport for 1:1 pixel=texture mapping
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
      gluOrtho2D(0.0, texWidth_X, 0.0, texHeight_X);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
```

```
  glViewport(0, 0, texWidth_X,texHeight_X);
}
void initCG()
{
      // Setup Cg
      cgSetErrorCallback(cgErrorCallback);
      cgContext = cgCreateContext();
      if(cgContext == NULL)
      {
  printf("Failed to create CGcontext\n");
  exit(-1);
      }
      checkError("cgCreateContext");
      vertexProfile = cgGLGetLatestProfile(CG_GL_VERTEX);
      fragmentProfile = cgGLGetLatestProfile(CG_GL_FRAGMENT);
      checkError("cgGLGetLatestProfile");
      // sanity check
      if   (vertexProfile==CG_PROFILE_UNKNOWN   ||   fragmentProfile   ==
CG_PROFILE_UNKNOWN)
      {
            logger->fatal("Unknown Cg Profile");
            exit (ERROR_CG);
      }
      cgGLSetOptimalOptions(vertexProfile);
      cgGLSetOptimalOptions(fragmentProfile);
      checkError("cgGLSetOptimalOptions");

      // create vertex programs
      vertexProgram1    =    cgCreateProgram    (cgContext,    CG_SOURCE,
vertexSource, vertexProfile,"vertexProgram",NULL);
      // create fragment programs
      fragmentProgram1    =    cgCreateProgram    (cgContext,    CG_SOURCE,
Jacobi_source, fragmentProfile, "Jacobi", NULL);

      // load programs
      cgGLLoadProgram (vertexProgram1);
      cgGLLoadProgram (fragmentProgram1);


      // and get parameter handles
      // from vertex program
      matrixParam                  =cgGetNamedParameter(vertexProgram1,
"ModelViewMatrix");
      // from the different fragment programs
      xParam1 = cgGetNamedParameter (fragmentProgram1, "textureX");
      bParam1 = cgGetNamedParameter (fragmentProgram1, "textureB");
      Ybus_imagParam1    =    cgGetNamedParameter    (fragmentProgram1,
"textureYbus_imag");
      Ybus_imag_ddinvParam1   =   cgGetNamedParameter   (fragmentProgram1,
"textureYbus_imag_ddinv");
      I_imagParam1     =     cgGetNamedParameter     (fragmentProgram1,
"textureI_imag");
      J_imagParam1     =     cgGetNamedParameter     (fragmentProgram1,
"textureJ_imag");
      Ybus_imag_ddParam1    =    cgGetNamedParameter    (fragmentProgram1,
"textureYbus_imag_dd");
```

```
	checkError("Parameters");
}

/*
 * Sets up GLUT, creates window.
 */
void initGLUT(int argc, char **argv)
{
	glutInit ( &argc, argv );
	windowHandle = glutCreateWindow("Jacobi iteration");
}

/*
 * Sets up GLEW, performs sanity check for required extensions.
 */
void initGLEW (void)
{
	int err = glewInit();
	if (GLEW_OK != err)
	{
		logger->fatal((char*)glewGetErrorString(err));
		exit(ERROR_GLEW);
	}
}



void printInfo()
{
	std::string s = "Problem size: N=";
	s.append (int2string(N));
	s.append(", tex=");
	s.append(int2string(texWidth_X));
	s.append("x");
	s.append(int2string(texHeight_X));
	logger->info(s);
	s = "Iterations: ";
	s.append(int2string(numIterations));
	logger->info(s);
}

void SWAP(void) {
  if (DESTINATION_BUFFER == 0) {
		DESTINATION_BUFFER = 1;
		SOURCE_BUFFER = 0;
  } else {
		DESTINATION_BUFFER = 0;
		SOURCE_BUFFER = 1;
  }
}
```

```
/*
 * utility function to enable texture parameter for Cg
 */
void enableTexture (GLuint texARB, GLuint texID, CGparameter param)
{
      glActiveTexture(texARB);
      glBindTexture(textureParameters.texTarget,texID);
      cgGLSetTextureParameter(param,texID);
      cgGLEnableTextureParameter(param);
}

/*
 * ping pong render pass (inner texels)
 */
void PerformComputation ()
{
      rgba* data=(rgba*)malloc(N*sizeof(rgba));
      glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
attachmentpoints[DESTINATION_BUFFER],       textureParameters.texTarget,
xTexID[DESTINATION_BUFFER], 0);
  glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
attachmentpoints[SOURCE_BUFFER],            textureParameters.texTarget,
xTexID[SOURCE_BUFFER], 0);
      cgGLEnableProfile(vertexProfile);
      cgGLEnableProfile(fragmentProfile);
      cgGLBindProgram(vertexProgram1);
      cgGLBindProgram(fragmentProgram1);

      // tell the shader to use the correct other textures
      enableTexture(GL_TEXTURE2_ARB, bTexID, bParam1);
      enableTexture(GL_TEXTURE3_ARB,Ybus_imagTexID, Ybus_imagParam1);
      enableTexture(GL_TEXTURE4_ARB,Ybus_imag_ddinvTexID,
Ybus_imag_ddinvParam1);
      enableTexture(GL_TEXTURE5_ARB,I_imagTexID,I_imagParam1);
      enableTexture(GL_TEXTURE6_ARB,J_imagTexID,J_imagParam1);
      enableTexture(GL_TEXTURE7_ARB,Ybus_imag_ddTexID,Ybus_imag_ddParam
1);


      cgGLSetStateMatrixParameter(matrixParam,CG_GL_MODELVIEW_PROJECTIO
N_MATRIX,CG_GL_MATRIX_IDENTITY);
      glFinish();
      //QueryPerformanceCounter(&start_ticks);




      for (int i=0; i<numIterations; i++)
      {
      glDrawBuffer(attachmentpoints[DESTINATION_BUFFER]);
      cgGLSetTextureParameter(xParam1, xTexID[SOURCE_BUFFER]);
      cgGLEnableTextureParameter(xParam1);
      // and render viewport-sized quad to make sure the fragment
program gets
      // executed for every pixel / texel we stored data in
```

```
	glBegin(GL_QUADS);
		glMultiTexCoord2fARB(GL_TEXTURE0_ARB,0.0, 0.0);
		glMultiTexCoord2fARB(GL_TEXTURE1_ARB,0.0, 0.0);
		glMultiTexCoord2fARB(GL_TEXTURE2_ARB,0.0, 0.0);
		glMultiTexCoord2fARB(GL_TEXTURE3_ARB,0.0, 0.0);
		glMultiTexCoord2fARB(GL_TEXTURE4_ARB,0.0, 0.0);
		glMultiTexCoord2fARB(GL_TEXTURE5_ARB,0.0, 0.0);
		glMultiTexCoord2fARB(GL_TEXTURE6_ARB,0.0, 0.0);
		glMultiTexCoord2fARB(GL_TEXTURE7_ARB,0.0, 0.0);


		glVertex2f(0.0,0.0);


		glMultiTexCoord2fARB(GL_TEXTURE0_ARB,texWidth_X, 0.0);
		glMultiTexCoord2fARB(GL_TEXTURE1_ARB,texWidth_X, 0.0);
		glMultiTexCoord2fARB(GL_TEXTURE2_ARB,texWidth_X, 0.0);
		glMultiTexCoord2fARB(GL_TEXTURE3_ARB,texWidth_Ybus, 0.0);
		glMultiTexCoord2fARB(GL_TEXTURE4_ARB,texWidth_X, 0.0);
		glMultiTexCoord2fARB(GL_TEXTURE5_ARB,texWidth_Ybus, 0.0);
		glMultiTexCoord2fARB(GL_TEXTURE6_ARB,texWidth_X, 0.0);
		glMultiTexCoord2fARB(GL_TEXTURE7_ARB,texWidth_X, 0.0);

		glVertex2f(texWidth_X, 0.0);


		glMultiTexCoord2fARB(GL_TEXTURE0_ARB,texWidth_X,
texHeight_X);
		glMultiTexCoord2fARB(GL_TEXTURE1_ARB,texWidth_X,
texHeight_X);
		glMultiTexCoord2fARB(GL_TEXTURE2_ARB,texWidth_X,
texHeight_X);
		glMultiTexCoord2fARB(GL_TEXTURE3_ARB,texWidth_Ybus,
texHeight_Ybus);
		glMultiTexCoord2fARB(GL_TEXTURE4_ARB,texWidth_X,
texHeight_X);
		glMultiTexCoord2fARB(GL_TEXTURE5_ARB,texWidth_Ybus,
texHeight_Ybus);
		glMultiTexCoord2fARB(GL_TEXTURE6_ARB,texWidth_X,
texHeight_X);
		glMultiTexCoord2fARB(GL_TEXTURE7_ARB,texWidth_X,
texHeight_X);

		glVertex2f(texWidth_X, texHeight_X);


		glMultiTexCoord2fARB(GL_TEXTURE0_ARB,0.0, texHeight_X);
		glMultiTexCoord2fARB(GL_TEXTURE1_ARB,0.0, texHeight_X);
		glMultiTexCoord2fARB(GL_TEXTURE2_ARB,0.0, texHeight_X);
		glMultiTexCoord2fARB(GL_TEXTURE3_ARB,0.0, texHeight_Ybus);
		glMultiTexCoord2fARB(GL_TEXTURE4_ARB,0.0, texHeight_X);
		glMultiTexCoord2fARB(GL_TEXTURE5_ARB,0.0, texHeight_Ybus);
		glMultiTexCoord2fARB(GL_TEXTURE6_ARB,0.0, texHeight_X);
		glMultiTexCoord2fARB(GL_TEXTURE7_ARB,0.0, texHeight_X);

		glVertex2f(0.0, texHeight_X);
	glEnd();
```

```
  transferFromTexture(data,texWidth_X,texHeight_X);

      SWAP();


      }

      glFinish();


}




 //* main routine

int  main(int argc, char **argv)
 {
       //int *err = (int *)x;
  //if (err == NULL) return 0;

  // Call application and library initialization. Perform this
  // initialization before calling any API functions or
  // Compiler-generated libraries.
  if (!mclInitializeApplication(NULL,0))
  {
  std::cerr << "could not initialize application properly"
  << std::endl;
      //*err = -1;
  //return x;
  }
  if( !gpudclibInitialize() )
  {
  std::cerr << "could not initialize library properly"
  << std::endl;
      //*err = -1;
  }
  else
  {
  try
  {
                //*define input parameters*//
           mwArray casename_in("case300.m");
           mwArray mpoption_in("mpoption");
           mwArray fname_in("case300_solved");
           mwArray casedata_in("case300_data");


           createAllTextureParameters();
           rgba* data=(rgba*)malloc(N*sizeof(rgba));

           // declaring variable for transferring data to C++ for GPU
```

```
            float VoltageAngleMatlab[N_actual];
            float dataPsch_g[N_actual];
            float dataYbus_real_g[N_actual*N_actual];
            float dataYbus_imag_g[N_actual*N_actual];
            float dataPYbus_imag_g[texHeight_Ybus*texWidth_Ybus];
            float dataPI_imag_g[texHeight_Ybus*texWidth_Ybus];
            float dataPJ_imag_g[texHeight_Ybus*texWidth_Ybus];
            float dataVoltageMag_g[N_actual];
            float dataVoltageAngle_g[N_actual];
            float dataPYbus_imag_dd_g[N];
            float dataPYbus_imag_ddinv_g[N];
            float VoltageAngleOut[N_actual][4];

            //declaring variable for transferring data to C++ for CPU

            //float VoltageAngleOut_CPU[N_actual];
            float dataVoltageAngleUpdate_CPU[N_actual];
            //float
VoltageAngleOut_CPU[N_actual][NumberOfCPUContingencies];
            float VoltageAngleOut_CPU[N_actual];
            float temp_update[N_actual];









      // specifying simulation parameters
            textureParameters = rect_nv_rgba_32;




      // start logger
      logger = Logger::getLogger(LOGFILENAME);
      logger->info("Log file for Jacobi: 9 bands, v, luminance");




float* Pcalc = (float*)malloc(N_actual*sizeof(float));
float* dataPmiss = (float*)malloc(N_actual*sizeof(float));


/* and output parameters to be passed to the library functions */
            mwArray Va_out;
            mwArray dataPsch_out;
            mwArray dataYbus_imag_out;
            mwArray dataYbus_real_out;
```

```
            mwArray dataPYbus_imag_out;
            mwArray dataPI_imag_out;
            mwArray dataPJ_imag_out;
            mwArray dataPYbus_imag_dd_out;
            mwArray dataPYbus_imag_ddinv_out;
            mwArray dataPmiss_out;
            mwArray dataVoltageMag_out;
            mwArray dataVoltageAngle_out;
```

```
  /* Call the mclInitializeApplication routine. Make sure that the
application
 *  was initialized properly by checking the return status. This
initialization
 * has to be done before calling any MATLAB API's or MATLAB Compiler
generated
 * shared library functions.  */
gpudc(11,Va_out,dataPsch_out,dataYbus_imag_out,dataYbus_real_out,dataPY
bus_imag_out,dataPI_imag_out,dataPJ_imag_out,dataPYbus_imag_dd_out,data
PYbus_imag_ddinv_out,dataVoltageAngle_out,dataVoltageMag_out,casename_i
n,mpoption_in,fname_in,casedata_in);

       //transferring data to C++
       Va_out.GetData(VoltageAngleMatlab,N_actual);
       dataPsch_out.GetData(dataPsch_g,N_actual);
       dataYbus_imag_out.GetData(dataYbus_imag_g,N_actual*N_actual);
       dataYbus_real_out.GetData(dataYbus_real_g,N_actual*N_actual);

dataPYbus_imag_out.GetData(dataPYbus_imag_g,texHeight_Ybus*texWidth_Ybu
s);

dataPI_imag_out.GetData(dataPI_imag_g,texHeight_Ybus*texWidth_Ybus);

dataPJ_imag_out.GetData(dataPJ_imag_g,texHeight_Ybus*texWidth_Ybus);
       dataPYbus_imag_dd_out.GetData(dataPYbus_imag_dd_g,N);
       dataPYbus_imag_ddinv_out.GetData(dataPYbus_imag_ddinv_g,N);
       //dataPmiss_out.GetData(dataPmiss_g,N);
 dataVoltageMag_out.GetData(dataVoltageMag_g,N_actual);
       dataVoltageAngle_out.GetData(dataVoltageAngle_g,N_actual);


// init GLUT, GLEW, RT, Cg
glFinish();
start=clock();
      initGLUT(argc, argv);
      checkError("GLUT INIT");
      initGLEW();
      checkError("GLEW INIT");
      initFBO();
      createTextures();
      initCG();
```

```
      checkError("CG INIT");

end = clock();
total_init = (end-start)/CLOCKS_PER_SEC;



////////////////////////////////////////////////////////////////////////
//
//STARTING GPU BASED CONTINGENCY
////////////////////////////////////////////////////////////////////////
//

glFinish();
start=clock();
for(int
currentcontingency=0;currentcontingency<NumberOfGPUContingencies;curren
tcontingency++)
{

for (int i=0;i<texHeight_Ybus;i++)
{
      dataYbus_imag_dd[i][0]=dataPYbus_imag_dd_g[i];
      dataYbus_imag_dd[i][1]=dataPYbus_imag_dd_g[i];
      dataYbus_imag_dd[i][2]=dataPYbus_imag_dd_g[i];
      dataYbus_imag_dd[i][3]=dataPYbus_imag_dd_g[i];

      dataYbus_imag_ddinv[i][0]=dataPYbus_imag_ddinv_g[i];
      dataYbus_imag_ddinv[i][1]=dataPYbus_imag_ddinv_g[i];
      dataYbus_imag_ddinv[i][2]=dataPYbus_imag_ddinv_g[i];
      dataYbus_imag_ddinv[i][3]=dataPYbus_imag_ddinv_g[i];

      dataUpdate[i][0]=0.0;
      dataUpdate[i][1]=0.0;
      dataUpdate[i][2]=0.0;
      dataUpdate[i][3]=0.0;

      dataMiss[i][0]=0.0;
      dataMiss[i][1]=0.0;
      dataMiss[i][2]=0.0;
      dataMiss[i][3]=0.0;




      for(int j=0;j<texWidth_Ybus;j++)
      {


            dataYbus[texWidth_Ybus*i+j][0]                              =
dataPYbus_imag_g[i+j*texHeight_Ybus];
            dataYbus[texWidth_Ybus*i+j][1]                              =
dataPYbus_imag_g[i+j*texHeight_Ybus];
            dataYbus[texWidth_Ybus*i+j][2]                              =
dataPYbus_imag_g[i+j*texHeight_Ybus];
```

```
            dataYbus[texWidth_Ybus*i+j][3]                                  =
dataPYbus_imag_g[i+j*texHeight_Ybus];


      dataI_imag[texWidth_Ybus*i+j][0]=dataPI_imag_g[i+j*texHeight_Ybus
];

      dataI_imag[texWidth_Ybus*i+j][1]=dataPI_imag_g[i+j*texHeight_Ybus
];
            dataI_imag[texWidth_Ybus*i+j][2]                                =
dataPI_imag_g[i+j*texHeight_Ybus];
            dataI_imag[texWidth_Ybus*i+j][3]                                =
dataPI_imag_g[i+j*texHeight_Ybus];


      dataJ_imag[texWidth_Ybus*i+j][0]=dataPJ_imag_g[i+j*texHeight_Ybus
];

      dataJ_imag[texWidth_Ybus*i+j][1]=dataPJ_imag_g[i+j*texHeight_Ybus
];
            dataJ_imag[texWidth_Ybus*i+j][2]                                =
dataPJ_imag_g[i+j*texHeight_Ybus];
            dataJ_imag[texWidth_Ybus*i+j][3]                                =
dataPJ_imag_g[i+j*texHeight_Ybus];


      }
}




//calculating mismatch
for(int i=0;i<N_actual;i++)
      {
                  Pcalc[i]=0;

                  for(int n=0;n<N_actual;n++)
                  {

                        float          angle2=sin(dataVoltageAngle_g[n]-
dataVoltageAngle_g[i]);
                        float
value1=dataYbus_imag_g[i+N_actual*n]*angle2;
                        Pcalc[i]=Pcalc[i]+value1;

                  }
                  dataPmiss[i]=dataPsch_g[i]-Pcalc[i];
                  VoltageAngleOut[i][0]=dataVoltageAngle_g[i];
                  VoltageAngleOut[i][1]=dataVoltageAngle_g[i];
                  VoltageAngleOut[i][2]=dataVoltageAngle_g[i];
                  VoltageAngleOut[i][3]=dataVoltageAngle_g[i];

      //cout<<"Pmiss["<<i<<"]="<<dataPmiss[i]<<'\t'<<"PScheduled["<<i<<
```

```
"]="<<dataPsch_g[i]<<'\t'<<"Angle["<<i<<"]="<<(dataVoltageAngle_g[i]*18
0)/pi<<endl;

	}


	for (int i=0;i<N_actual;i++)
	{
		dataMiss[i+(N-N_actual)][0]=dataPmiss[i];
		dataMiss[i+(N-N_actual)][1]=dataPmiss[i];
		dataMiss[i+(N-N_actual)][2]=dataPmiss[i];
		dataMiss[i+(N-N_actual)][3]=dataPmiss[i];

	}

	////////////////////////////////////////////////////////////////
///////
	// data upload
	////////////////////////////////////////////////////////////////
/////


	SOURCE_BUFFER = 0;
	DESTINATION_BUFFER=1;
	transferToTexture(dataMiss,bTexID,texWidth_X,texHeight_X);
	transferToTexture(dataUpdate,xTexID[SOURCE_BUFFER],texWidth_X,tex
Height_X);
	transferToTexture(dataYbus,Ybus_imagTexID,texWidth_Ybus,texHeight
_Ybus);
	transferToTexture(dataYbus_imag_ddinv,Ybus_imag_ddinvTexID,texWid
th_X,texHeight_X);
	transferToTexture(dataI_imag,I_imagTexID,texWidth_Ybus,texHeight_
Ybus);
	transferToTexture(dataJ_imag,J_imagTexID,texWidth_Ybus,texHeight_
Ybus);
	transferToTexture(dataYbus_imag_dd,Ybus_imag_ddTexID,texWidth_X,t
exHeight_X);


	////////////////////////////////////////////////////////////////
/////
	// computing
	////////////////////////////////////////////////////////////////
////


	PerformComputation();




////////////////////////////////////////////////////////////////////////
///
	// downloading
	////////////////////////////////////////////////////////////////
////////
```

```
	/*glFinish();
	start=clock();    */

	transferFromTexture(data,texWidth_X,texHeight_X);

	/*end = clock();
	total_download = total_download + ((end-start)/CLOCKS_PER_SEC);*/

	/*for(int i=0;i<N_actual;i++)
	{
		cout<<"VoltageAngleUpadte["<<i+1<<"]="<<data[i+2][0]<<'\t';


	}*/

	for(int i=0;i<N_actual;i++)
	{
		VoltageAngleOut[i][0]=VoltageAngleOut[i][0]+data[i+(N-
N_actual)][0];
		VoltageAngleOut[i][1]=VoltageAngleOut[i][1]+data[i+(N-
N_actual)][1];
		VoltageAngleOut[i][2]=VoltageAngleOut[i][2]+data[i+(N-
N_actual)][2];
		VoltageAngleOut[i][3]=VoltageAngleOut[i][3]+data[i+(N-
N_actual)][3];
	}

for(int i=0;i<N_actual;i++)
	{

		VoltageAngleOut[i][0]=(VoltageAngleOut[i][0]*180)/pi;
		VoltageAngleOut[i][1]=(VoltageAngleOut[i][1]*180)/pi;
		VoltageAngleOut[i][2]=(VoltageAngleOut[i][2]*180)/pi;
		VoltageAngleOut[i][3]=(VoltageAngleOut[i][3]*180)/pi;

	//cout<<VoltageAngleOut[i][0]<<'\t'<<VoltageAngleOut[i][1]<<'\t'<
<VoltageAngleOut[i][2]<<'\t'<<VoltageAngleOut[i][3]<<'\t'<<endl;
	}


}
end = clock();
total_comp = total_comp + ((end-start)/CLOCKS_PER_SEC);

// disabling profiles
cgGLDisableProfile(vertexProfile);
cgGLDisableProfile(fragmentProfile);

////////////////////////////////////////////////////////////////////////
//////////////////////
// END GPU BASED CONTINGENCY
////////////////////////////////////////////////////////////////////////
//////////////////////
```

```
////////////////////////////////////////////////////////////////////////
//////////////////////////
//STARTING CPU BASED CONTINGENCY
////////////////////////////////////////////////////////////////////////
//////////////////////////
glFinish();
start=clock();

for(                                                                    int
currentCPUcontingency=0;currentCPUcontingency<NumberOfCPUContingencies;
currentCPUcontingency++)
{

//calculating mismatch
for(int i=0;i<N_actual;i++)
      {
                  Pcalc[i]=0;

                  for(int n=0;n<N_actual;n++)
                  {

                        float          angle2=sin(dataVoltageAngle_g[n]-
dataVoltageAngle_g[i]);

                        float
value1=dataYbus_imag_g[i+N_actual*n]*angle2;

                        Pcalc[i]=Pcalc[i]+value1;

                  }
                  dataPmiss[i]=dataPsch_g[i]-Pcalc[i];
                  dataVoltageAngleUpdate_CPU[i]=0;


      //cout<<"Pmiss["<<i<<"]="<<dataPmiss[i]<<'\t'<<"PScheduled["<<i<<
"]="<<dataPsch_g[i]<<'\t'<<"Angle["<<i<<"]="<<(dataVoltageAngle_g[i]*18
0)/pi<<endl;
                  //cout<<endl;


      }

// Computing updates

  for                                                                   (int
currentCPUIteration=0;currentCPUIteration<numIterations;currentCPUItera
tion++)
      {

  for (int i=0;i<N_actual;i++)
            {
    float out=0;

    for (int j=0;j<N_actual;j++)
                  {
```

```
out=out+dataYbus_imag_g[i+N_actual*j]*dataVoltageAngleUpdate_CPU[j];

                  }



                  if(dataYbus_imag_g[i*(N_actual+1)]!=0)
                  {


      temp_update[i]=(1/dataYbus_imag_g[i*(N_actual+1)])*(dataPmiss[i]-
out);
                  }
                  else
                  {
                        temp_update[i]=0;
                  }


                  dataVoltageAngleUpdate_CPU[i]=
dataVoltageAngleUpdate_CPU[i]+temp_update[i];


      //VoltageAngleOut_CPU[i][currentCPUcontingency]=dataVoltageAngle_
g[i]+dataVoltageAngleUpdate_CPU[i];

      VoltageAngleOut_CPU[i]=dataVoltageAngle_g[i]+dataVoltageAngleUpda
te_CPU[i];
            }
      }
}
end = clock();
total_comp_CPU = total_comp_CPU + ((end-start)/CLOCKS_PER_SEC);

//for(int i=0;i<N_actual;i++)
//    {
//
      cout<<"GPU["<<i+1<<"]="<<data[i+2][0]<<'\t'<<"CPU["<<i+1<<"]="<<d
ataVoltageAngleUpdate_CPU[i]<<endl;
//
//
//    }

////////////////////////////////////////////////////
//END CPU BASED CONTINGENCY
////////////////////////////////////////////////////
//for(int i=0;i<N_actual;i++)
//    {
//
      cout<<"VoltageAngleOut_CPU["<<i+1<<"]="<<(VoltageAngleOut_CPU[i][
0]*180)/pi<<endl;
//
//
//    }

////////////////////////////////////////////////////////////////////
```

```
//do cleanup
////////////////////////////////////////////////////////////////////
      glDeleteFramebuffersEXT(1,&fb);
      glDeleteTextures (2, xTexID);
      glDeleteTextures (1, &bTexID);
      glDeleteTextures (1, &Ybus_imagTexID);
      glDeleteTextures (1, &Ybus_imag_ddinvTexID);

////////////////////////////////////////////////////
//PRINTING RESULTS
////////////////////////////////////////////////////

cout<<"*****************************************************************
*";
cout<<endl;
cout<<endl;
cout<<endl;
cout<< "Total Number of Internal Iterations="<<numIterations<<endl;
cout<<"total                number                of                GPU
contingencies="<<NumberOfGPUContingencies<<endl;
cout<<"total                number                of                CPU
contingencies="<<NumberOfCPUContingencies<<endl;
cout<< "GPU computation time="<<total_comp<<endl;
cout<< "CPU computation time="<<total_comp_CPU<<endl;
//cout<< "data download time="<<total_download<<endl;



////////////////////////////////////////////////////////////////////////
//////////////////////

}
  catch (const mwException& e)
  {
  std::cerr << e.what() << std::endl;
 // *err = -2;
  }
  catch (...)
  {
  std::cerr << "Unexpected error thrown" << std::endl;
 // *err = -3;
  }
  // Call the application and library termination routine
            gpudclibTerminate();
  }
#ifdef __APPLE_CC__
  mclSetExitCode(*err);
#endif
  mclTerminateApplication();

      return 0;


}
```

10 APPENDIX B: Vertex Shader Source Code

```
static char *vertexSource =
" void vertexProgram (in float4 inpos: POSITION, \n"
"   out float4 outpos: HPOS, \n"
"   const uniform float4x4 ModelViewMatrix) {\n "
// do viewspace transform (we have to, since the hardware doesn't any
more)
"  outpos = mul (ModelViewMatrix, inpos);\n"
" } \n ";
```

11 APPENDIX C: Fragment Shader Source Code

```
static char *Jacobi_source =
 "float4 Jacobi(in float4 screen : WPOS, \n"
 "uniform samplerRECT textureX,const uniform samplerRECT textureB,
const uniform samplerRECT textureYbus_imag, const uniform samplerRECT
textureYbus_imag_ddinv,const uniform samplerRECT textureI_imag,const
uniform samplerRECT textureJ_imag,const uniform samplerRECT
textureYbus_imag_dd):COLOR \n"


  "{\n "
  "  float4 OUT;\n"
  "  float texwidth=10;\n"
  "  float2 ocoords = screen.xy; \n"
  "  float2 Ycoords=float2(0.0,0.0); \n"
  "  float2 Ycoords_temp=float2(0.0,0.0); \n"
  "  float2 Xcoords=float2(0.0,0.0); \n"
  "  float4 y; \n"
  "  float4 x; \n"
  "  float4 i;\n"
  "  float4 b;\n"
  "  float4 invDD;\n"


  " Ycoords.x=0.5; \n"
  " Ycoords.y=((ocoords.x-0.5)+((ocoords.y-0.5)*texwidth))+0.5; \n"


  "for(float j=0.0;j<11.0;j++)"
  "{"
  "  Ycoords_temp=Ycoords+half2(j,0.0);\n"
  "  y = texRECT (textureYbus_imag, Ycoords_temp);\n "
  "  Xcoords.y = texRECT (textureI_imag, Ycoords_temp);\n"
  "  Xcoords.x=texRECT (textureJ_imag, Ycoords_temp);\n"
  "  x = texRECT (textureX, Xcoords);\n"
  "  OUT = OUT+x*y;\n"
  "}"


  "  x = texRECT (textureX, ocoords);\n"
  "  y = texRECT (textureYbus_imag_dd, ocoords);\n"
  "  OUT = OUT+x*y;\n"


  // now crank it up to proper jacobi
  // by updating x=x+(1/DD)(b-Ax)

  "  b = texRECT (textureB, ocoords);\n"
  "  invDD = texRECT (textureYbus_imag_ddinv, ocoords);\n"
  "  OUT = (b-OUT)*invDD + x;\n"
  //"  OUT = texRECT (textureX, ocoords);\n"
```

```
  //"  OUT = b;\n"
  "  return OUT;\n "
"}\n";
```